TEKTRONIX®

8002
μPROCESSOR LAB
9900
ASSEMBLER & EMULATOR
USER'S MANUAL
Opt. 4, 19 & 34

This manual supports TEKDOS Version 2.

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077
070-2417-00

Serial Number _____

First Printing JAN 1978

WARRANTY

The 8001/8002 μProcessor Lab System (including options) is warranted against defective materials and workmanship under normal use and service for a period of 90 days from date of initial shipment. CRTs found to be defective within 12 months from the date of shipment will be exchanged at no charge (this does not include installation).

On site warranty repairs is provided during normal working hours (for the 90-day period). Travel to the site is confined to those areas in which Tektronix states it has service facilities available for this product.

Tektronix shall be under no obligation to furnish warranty service if:

a. Attempts to install, repair, or service the equipment are made by personnel other than Tektronix service representatives.

b. Modifications are made to the hardware or software by personnel other than Tektronix service representatives.

c. Damage results from connecting the 8001/8002 μProcessor Lab System to incompatible equipment.

Specifications and price change privileges reserved.

Copyright © 1978 by Tektronix, Inc., Beaverton, Oregon. Printed in the United States of America. All rights reserved. Contents of this publication may not be reproduced in any form without permission of Tektronix, Inc.

U.S.A. and foreign Tektronix products covered by U.S. and foreign patents and /or patents pending.

DOCUMENTATION OVERVIEW

INTRODUCTION

The 8002 μProcessor Lab support documentation consists of two groups of manuals: user's manuals and service manuals. User's manuals explain the procedures required to operate the 8002 μProcessor Lab system and its peripheral devices. These manuals, identified by their gray covers, are a standard part of the system package.

Service manuals provide the information necessary to perform routine maintenance and to make minor repairs to system components. The hardware test manuals within this group provide detailed troubleshooting information beyond the scope of routine maintenance. Service manuals, identified by their blue covers, may be purchased as optional accessories.

USER MANUAL ORGANIZATION

The 8002 μProcessor Lab user's manuals are incorporated into a series of user support packages. Each package contains a three-ring binder, a manual, a reference card that summarizes the contents of the manual, and one or more flexible discs. Some discs are blank and others contain coded programs.

The contents of the user support packages at this writing are as follows:

8002 μProcessor Lab System User's Package

General purpose three-ring binder
8002 μProcessor Lab System User's Manual
8002 μProcessor Lab System Reference Card
Two blank flexible discs

Description

This package is a standard accessory to every 8002 μProcessor Lab System. The system user's manual is the fundamental documentation and explains how to use the 8002 operating system. The system reference card summarizes the contents of the system user's manual. The two blank flexible discs are provided so that back-up copies of software can be safely stored. A blank disc may also be used to store user-written programs.

8002 μPROCESSOR LAB ASSEMBLER & EMULATOR SUPPORT PACKAGES FOR MICROPROCESSOR OPTIONS

These packages support program development for each optional microprocessor. Each system disc contains the TEKDOS operating system and the appropriate TEKTRONIX Assembler. The manuals contain the details necessary to operate the applicable assembler and emulator modules. In conjunction with the system user's manual, each assembler and emulator user's manual provides complete information for microprocessor program development. Assembler and emulator reference cards summarize the assembler and emulator commands and serve as quick reference guides.

For 8080/8085 Microprocessor

Contents

General purpose three-ring binder

8002 μProcessor Lab Assembler & Emulator User's Manual for 8080/8085 Microprocessor

8002 μProcessor Lab 8080/8085 Assembler and Emulator Reference Card

8002 system disc for 8080/8085 Microprocessor

For 6800 Microprocessor

Contents

General purpose three-ring binder

8002 μProcessor Lab Assembler & Emulator User's Manual for 6800 Microprocessor

8002 μProcessor Lab 6800 Assembler & Emulator Reference Card

8002 system disc for 6800 Microprocessor

For Z80 Microprocessor

Contents

General purpose three-ring binder

8002 μProcessor Lab Assembler & Emulator User's Manual for Z80 Microprocessor

8002 μProcessor Lab Z80 Assembler & Emulator Reference Card

8002 system disc for Z80 Microprocessor

For 9900 Microprocessor

Contents

General purpose three-ring binder

8002 μProcessor Lab Assembler & Emulator User's Manual for 9900 Microprocessor

8002 μProcessor Lab 9900 Assembler & Emulator Reference Card

8002 system disc for 9900 Microprocessor

FUTURE USER SUPPORT PACKAGES

Each microprocessor development module to be introduced in the future will be accompanied by a support package similar to those described above.

SERVICE MANUALS

Service documentation for the 8002 μProcessor Lab consists of a main system service manual and supplementary service manuals for each plug-in module. The service manuals contain information for installing, servicing, and maintaining system components. The user will find diagrams and circuit descriptions, specifications, and parts lists. Detailed maintenance information facilitates all necessary cleaning, lubrication, calibration, and diagnostic troubleshooting.

The following service manuals are available:

8002 μProcessor Lab System Service Manual

System Processor
System Memory
Program Memory
Assembler Processor
System Communications
Debug and Front Panel I/O
Flexible Disc Unit

8002 μProcessor Lab 8080/8085 Emulator Processor Service Manual

8080/8085 Emulator Processor
8080/8085 Prototype Control Probe

8002 μProcessor Lab 6800 Emulator Processor Service Manual

6800 Emulator Processor
6800 Prototype Control Probe

8002 μProcessor Lab Z80 Emulator Processor Service Manual

Z80 Emulator Processor
Z80 Prototype Control Probe

8002 μProcessor Lab 9900 Emulator Processor Service Manual

9900 Emulator Processor

9900 Prototype Control Probe

8001/8002 μProcessor Lab Real-Time Prototype Analyzer System Service Manual

Data Acquisition Interface

P6451 Data Acquisition Probe

8002 μProcessor Lab 2704/2708 PROM Programmer Service Manual

Service Instructions

8002 μProcessor Lab 1702 PROM Programmer Service Manual

Service Instructions

8002 μProcessor Lab Maintenance Front Panel Instruction Manual

Operating Instructions

Service Instructions

8002 μProcessor Lab Hardware Test Manual

Contains support documentation necessary to troubleshoot the 8002 system effectively. The manual, together with diagnostic software and a test fixture, forms the 8002 Hardware Test Package.

ABOUT THIS MANUAL

This manual explains 8002 μProcessor Lab assembly, linking, and emulation procedures for 9900-based microcomputer development. The user should be familiar with hexadecimal and binary number systems, and with ASCII character code. It is especially helpful if your programming background includes assembly language experience.

The manual describes all TEKTRONIX 9900 Assembler features and procedures in detail. These include: the basic source module format; all assembler directives; macro capability; assembled listing and object module formats; and procedures for linking assembled object code. Assembler operating procedures are discussed in this manual and in the TEKTRONIX 8002 μProcessor Lab System User's Manual.

The closing sections provide a detailed description of emulation procedures specific to 9900-based microprocessor development, including 9900 service calls, debugging, and prototype control probe specifics.

The appendices contain essential summarized information and conversion tables. Appendix C is an alphabetical summary of 9900 assembly language instructions. Appendix F lists all error codes, messages, and their associated explanations.

Throughout this manual, zeros are slashed where needed for clarity.

CAUTION

The TEKTRONIX Assembler software in this manual is designed to support the Texas Instrument TMS9900 Microprocessor. Therefore, all references to the 9900 Microprocessor in this manual pertain to the TMS9900 Microprocessor.

Contents

Page

SECTION 1 TEKTRONIX 9900 ASSEMBLER INTRODUCTION

Assembler Input 1-1
Assembler Output 1-1

SECTION 2 ASSEMBLER SOURCE MODULE FORMAT

Introduction 2-1
9900 Symbolic Statement Format 2-1
The Label Field 2-3
The Operation Field 2-3
The Operand Field 2-4
The Comment Field 2-8
Using Symbols 2-9
Programmer-Defined Symbols 2-9
Pre-defined Symbols 2-10
Rules for Creating Symbols 2-10
Numeric Values 2-10
Scalar Values 2-10
Address Values 2-11
Notation Rules for Specifying Constants 2-11
Numeric Constants 2-11
String Constants 2-12
Null Strings 2-12
String to Numeric Conversion 2-13
Expressions Permitted in the Operand Field 2-13
Hierarchy of Expression Operators and Functions 2-16
Description of Expression Operators and Functions 2-16
Binary Arithmetic Operators 2-17
Unary Operators 2-18
Relational Operators 2-18
Numeric Comparisons 2-18
String Comparisons 2-19
String Concatenation 2-20
Functions 2-21
String Variables 2-24
SET Strings 2-25
String Text Substitution 2-26

Page

SECTION 3 STATEMENT SYNTAX CONVENTIONS

Introduction 3-1
Tektronix Assembler Statement Syntax 3-1
Use of Upper and Lower Case Letters and Punctuation 3-2
Blank Fields 3-2
Braces and Brackets 3-2
Trailing Dots 3-2
TEKDOS Statement Syntax 3-3
Command Name 3-3
Delimiters 3-3
Parameters 3-4
Braces and Brackets 3-4
Trailing Dots 3-4

SECTION 4 ASSEMBLER DIRECTIVES

Introduction 4-1
Listing Format Control Directives 4-3
LIST and NOLIST 4-4
General Listing Format Control Options 4-4
Macro Listing Format Control Options 4-5
Conventions for Listing Control 4-6
PAGE 4-8
SPACE 4-11
TITLE 4-14
STITLE 4-15
WARNING 4-17
Symbol Definition Directives 4-19
EQU 4-20
STRING 4-21
SET 4-23
Workspace Location Determination Directive 4-26
WPNT 4-26
Location Counter Control Directive 4-27
ORG 4-27
Data Storage Control Directives 4-30
BYTE 4-31
WORD 4-33
ASCII 4-35
BLOCK 4-38
Macro Definition Directives 4-39
MACRO 4-40
ENDM 4-42
REPEAT and ENDR 4-43
INCLUDE 4-45
Conditional Assembly Directives 4-47
IF, ELSE, and ENDIF 4-48
EXITM 4-51
Section Definition Directives 4-53
Relocation Option 4-54

Page

SECTION 4 ASSEMBLER DIRECTIVES (CONT)

SECTION 4-55
COMMON 4-57
RESERVE 4-59
RESUME 4-61
GLOBAL 4-63
NAME 4-65
Module Termination Directive 4-66
END 4-66

SECTION 5 MACROS

Introduction 5-1
Basic Macro Expansion Process 5-1
Macro Definition Directive 5-2
Macro Definition Directive Conventions 5-3
Macro Definition Block 5-3
Source Code Alteration 5-3
Additional Special Macro Definition Conventions 5-4
The @ Character 5-4
The # Character 5-5
The % Character 5-5
The ↑ or ∧ Character In Macro Definition 5-6
Macro Termination 5-6
Macro Calling 5-7
INCLUDE Directive Text Insertion 5-7
Text Substitution 5-8
Special Macro Calling Characters 5-9
The [ ] Construct 5-9
The ↑ or ∧ Character In Macro Calls 5-10
Additional Macro Argument Conventions 5-10
Examples 5-11
Conditional Assembly 5-15
Nesting 5-16
Conditional Macro Termination 5-16
EXAMPLES 5-16
IF-ENDIF Blocks 5-16
REPEAT-ENDR Blocks 5-18
Macro Expansion Summary 5-20

SECTION 6 ASSEMBLER OPERATING PROCEDURES

Introduction 6-1
Purpose 6-1
Explanation 6-2
Assembly Completion 6-3

Page

SECTION 7 ASSEMBLER LISTING FORMAT

Introduction 7-1
The Assembler Listing 7-1
Headings 7-2
The Listing Line 7-3
Error Response 7-5
The Symbol Table 7-6

SECTION 8 ASSEMBLER OBJECT MODULE FORMAT

Introduction 8-1
Program Loading and Execution 8-1
LOAD 8-2
GO 8-3

SECTION 9 THE LINKER

Introduction 9-1
Linker Invocation 9-2
Program Sections 9-3
Section Attributes 9-3
Linker Invocation 9-5
Simple Invocation 9-5
Interactive Command Invocation 9-6
Command File Invocation 9-7
Commands 9-8
Memory Location 9-11
Memory Allocation of Sections 9-11
ENDREL 9-12
Linker Output 9-12
Listing File 9-12
Error Messages 9-13
Map 9-13
Symbol List 9-14
Linker Statistics 9-15
The Load File 9-15
Errors and Error Messages 9-16
Error Messages and Explanations 9-16
Command Processing Errors 9-19
Extraneous Information Ignored 9-19
Illegal Command 9-19
Syntax Error 9-19
Indirect File Depth Exceeded 9-19
Invalid File Name 9-19
Invalid Range Specified 9-19

SECTION 10 9900 SERVICE CALLS

Introduction 10-1
The 9900 SVC Compare Word Operation 10-2

Page

SECTION 11 9900 DEBUGGING

Introduction 11-1
TRACE 11-2
The Trace Modes 11-3
The Trace Line 11-3
Debug Error Responses 11-4
Trace Line Termination 11-5
DSTAT 11-7
SET 11-11

SECTION 12 PROTOTYPE CONTROL PROBE

Introduction 12-1
Description and Installation 12-1
Operation 12-5

APPENDIX A SOURCE MODULE CHARACTER SET A-1

APPENDIX B ASSEMBLER DIRECTIVES B-1

Assembler Directive Syntax B-3

APPENDIX C SUMMARY OF 9900 INSTRUCTIONS C-1

Data Transfer Instructions C-7
Arithmetic Instructions C-7
Comparison Instructions C-9
Logical Instructions C-11
Shift and Rotate Instructions C-12
Control Transfer Instructions C-13
Communications Register Unit (CRU) Instructions C-15
Interrupt Control Instructions C-18

APPENDIX D SERVICE CALL FUNCTION CODES D-1

APPENDIX E HEXIDECIMAL CONVERSION TABLES E-1

ASCII Code Conversion Table E-1
Decimal-Hexadecimal-Binary Equivalents E-2
Hexadecimal Addition Table E-3
Hexadecimal Multiplication Table E-4

APPENDIX F ASSEMBLER ERROR CODES F-1

APPENDIX G RESERVED WORDS G-1

2417-1

Fig. 1-1. The 8002 μProcessor Lab System with optional CT8100 CRT Terminal and 9900 Prototype Control Probe.

Section 1
TEKTRONIX 9900 ASSEMBLER INTRODUCTION

ASSEMBLER INPUT

The TEKTRONIX 9900 Assembler translates user-written programs into executable binary format. The user's program must be written in 9900 symbolic notation (assembly language), and becomes the source module for assembler operation. User-written programs can be entered into disc files with the text editor program, using procedures described in the TEKTRONIX 8002 μProcessor Lab System User's Manual. If the source module is contained in more than one flexible disc file, each file name must be specified by assemble command (ASM) parameters.

All valid input devices can originate assembler input. The assembler reads the source module twice, once for each pass. When it encounters an END directive or reads the end of the last file during the first pass, the assembler begins the second pass and starts assembly.

ASSEMBLER OUTPUT

Assembler output comprises an object module, program listings, and appropriate information messages. The object module contains executable binary instructions and data constants translated from the source module. The entire object file must be linked and then loaded into program memory in order to execute the translated user program on the 9900 Emulator Processor.

Program listings produced by the assembler are composed of line numbers, the generated object code, and the source code as entered in the source module. Wherever an error is detected, an error code is printed on the display device and to the listing to specify the nature of the problem.

Following the source code listing, a symbol table alphabetically lists all symbols entered in the program. The table also gives the hexadecimal value of each symbol and indicates undefined symbols. Below the symbol table, a message indicates the number of source lines, the number of assembled lines, the number of bytes available, and the number of errors and undefined symbols.

To transfer the listing and object file to a disc, enter output file names as ASM command parameters. To transfer assembler listing and object files to an output device (such as a line printer) instead of a file, specify the name of the device as the ASM command parameter.

The TEKTRONIX Assembler makes two passes through the source module. The first pass determines the number of storage bytes required for each statement, and assigns a starting address value for the first byte of each statement line. The location counter, set to zero before the first pass begins, advances after each statement is read. This action effectively generates the starting address for each statement. The symbol table is also constructed during the first pass. During the second pass, the source module and the symbol table are used to generate the object module and the listings.

After assembly completion, each line containing an error is output to the display device, with an error code specifying the nature of the error. Below all error displays, a message indicates the number of source lines, the number of assembled lines, the number of bytes available, and the number of any errors or undefined symbols. If an irrecoverable error prevents assembly completion, the program aborts and an error code indicates the cause.

Section 2

ASSEMBLER SOURCE MODULE FORMAT

INTRODUCTION

Symbolic 9900 instructions, assembler directives, macro calls, and explanatory comments form the source module. Each 9900 source module statement must be entered according to the TEKTRONIX 9900 Assembler format. When translated by the assembler, the source module becomes the object module to be executed.

Three types of source module statements may be used:

1. 9900 symbolic instructions,
2. assembler directives, and
3. macro calls.

9900 SYMBOLIC STATEMENT FORMAT

Each source module line may contain up to 128 characters, and is terminated by a carriage return. Allowable source module characters are detailed in Appendix A. Blank lines can be used to improve readability of the source module listing. The blank lines do not affect the translated program.

Each 9900 instruction, assembler directive, or macro call consists of four fields: the label field, the operation field, the operand field, and the comment field. During program assembly, each 9900 source module instruction is translated by the assembler into one, two, or three words of code in the object module. The length depends upon the instruction type, and the number and type of operands required.

The label field, when used, must begin in the first-character position of a line. The operation and operand fields must begin anywhere after the first-character position and end in any line character position within the 128-character range. The comment field may begin in any line character position and must end within the 128-character range. Field sequence may not be changed, however; and the correct order can only be as follows:

LABEL OPERATION OPERAND COMMENT

Throughout this manual, this field sequencing format is shown above each source line to illustrate proper assembler source line formatting.

Readability is improved when each field in the source module begins at a constant position within the line. This columnar format can be easily implemented by using the tab setting function to define field starting positions. Fig. 2-1 is an example of a properly formatted source module.

```
LABEL   OPERATION   OPERAND     COMMENT
        STRING      S1 (8Ø)     ;DEFINE STRING VARIABLE S1 WITH 8Ø
                                ;CHARACTER MAXIMUM
L1      EQU         3           ;DEFINE CONSTANT SYMBOL L1 TO EQUAL 3
L2      SET         4           ;DEFINE VARIABLE SYMBOL L2 TO EQUAL 4
        ORG         11ØØH       ;STARTS OBJECT CODE OF NEXT INSTRUCTION
                                ;AT 11ØØH
        MOV         R1,R2       ;LOAD REG. 2 WITH CONTENTS OF REG. 1
        END                     ;END OF PROGRAM
```

Fig. 2-1. Properly Formatted 9900 Program.

A general description of the characteristics of each source module field follows. The entire 9900 instruction set is listed in Appendix C. The TEKTRONIX Assembler directives are described in Section 4 and listed in Appendix B. Macro calls are described in Section 5.

THE LABEL FIELD

Labels may be used in all 9900 instructions, macro calls, and assembler directives. Every label must be unique within each source module. Duplicate labels prevent proper program execution and cause an error code to appear on the display device and in the listing. The label field, when used, must start in the first-character position of the line. A blank or tab terminates the label field; therefore, imbedded blanks or tabs are not permitted within the field.

Labels represent addresses associated with locations in a source module. The EQU and SET directives are the only statements requiring label usage. In all other directives, label usage is optional. EQU and SET directives always equate the required label to the constant or expression value in the operand field. The SET directive allows the assigned symbol value be modified; the EQU directive does not. For all other directives, the label meaning is dependent upon the particular directive. Generally, the label translates to the memory address of data or a data constant value. A label in a 9900 instruction translates to the address of the first byte of the instruction.

ORG, WPNT, and BLOCK directives must contain constants or operand symbols that have already been defined. Operands in all other directives may reference label symbols that are defined in later statements.

THE OPERATION FIELD

The operation field contains the mnemonic operation code for a 9900 symbolic instruction, an assembler directive, or a macro call. The mnemonic specifies the operation or function to be performed at program execution time, or by the assembler during program translation and assembly. An instruction specifies the object code to be generated and the action to be performed on any operands that follow. An assembler directive specifies certain actions to be performed during assembly and might not generate any object code. The macro call specifies the macro definition block to be expanded.

The operation field begins after the label field is terminated. If the label is omitted, the operation field may begin anywhere after the first-character position in the line. The operation field is terminated by one or more spaces, by a tab or carriage return, or by a semicolon indicating the start of a comment field.

If the operation field does not contain a 9900 instruction, an assembler directive, or a macro call, the assembler rejects the entire statement and prints an error code. Six bytes of zero value are generated by the assembler to fill the area where a valid instruction would otherwise have been stored.

THE OPERAND FIELD

The operand field specifies values or locations required for the given assembler directive, instruction, or macro call. The operand field, if present, begins after the operation field is terminated. Spaces may be used in the operand field. Two or more operands are separated by commas. The field is terminated by a carriage return, or by a semicolon indicating the start of a comment field.

The operation code (appearing in the operation field) determines the type and number of items required for the operand field. If more than one item is required, the sequence of item appearance is determined by the operation code.

Operands required for macro calls and assembler directives are discussed in Sections 4 and 5, and summarized in Appendix B.

Eleven types of information are permitted in the instruction operand field. Each instruction determines the operand types and their proper sequence. Refer to Appendix C for a summary of 9900 instruction requirements.

The following list defines the eleven operand item types and their required syntax for 9900 instructions:

OPERAND TYPE	OPERAND SYNTAX
1) A workspace register which contains the operand.	RØ . . R15 expression
2) A workspace register containing the memory address of the operand. (Register indirect addressing)	*RØ . . *R15 *(expression)
3) A workspace register containing the memory address of the operand. After the address is obtained from the register, the register contents are incremented. (Register indirect autoincrement addressing.)	*RØ+ . . *R15+ *(expression)+
4) A 16-bit memory address within the range 0 to 65,535, containing the operand data.	expression @expression
5) A 16-bit indexed memory address, specified by a 16-bit absolute memory address plus the contents of the specified workspace register. This computed address contains the operand.	expression (RØ) @expression (RØ) . . expression (R15) @expression (R15) expression(expression) @expression(expression)
6) A 16-bit memory address within the range 0 to 65,535, which is the destination of control transfer, and is stored in the object module as an 8-bit PC-relative address within the range −128 to 127 words.	expression

OPERAND TYPE	OPERAND SYNTAX
7) An 8-bit CRU displacement address within the range −128 to 127, which is added to the base address in R12 bits 1 to 12 to form the effective CRU address.	expression
8) A 16-bit data or address constant within the range 0 to 65,535. An immediate value.	expression
9) A 4-bit Extended Operation (XOP) vector number within the range 0 to 15.	expression
10) A 4-bit value indicating the number of bits to transfer to or from the CRU.	expression
11) A 4-bit value indicating the number of bit position to shift or rotate the source operand. The operand is optional. If it is absent a value of Ø is assumed.	expression

Several 9900 instructions may operate on data in one or two of the sixteen 16-bit workspace registers. The operand field for these instructions may contain the register symbol or register value for each register involved. Register symbols may not be terms of an expression. Pre-defined register values are as follows:

REGISTER SYMBOL	REGISTER VALUE
RØ	Ø
R1	1
R2	2
R3	3
R4	4
R5	5
R6	6
R7	7
R8	8
R9	9
RIØ	1Ø
R11	11
R12	12
R13	13
R14	14
R15	15

The $ is used within operands to symbolize the first byte of the statement in which it appears. The effect of $ usage is equivalent to using a label in that statement. When using the $ to reference addresses, consult Appendix C for the number of bytes in each instruction. The two instruction sequences that follow are equivalent.

```
   LABEL     OPERATION     OPERAND     COMMENT
1) TIMER     DEC           RØ          ;DECREMENT RØ REGISTER,
                                       ;LABEL INSTRUCTION TIMER
             JNE           TIMER       ;JUMP BACK IF RØ NON-ZERO

2)           DEC           RØ          ;DECREMENT RØ REGISTER
             JNE           $–2         ;JUMP BACK IF RØ NON-ZERO
```

The $ represents the address of the first byte in the JNE instruction. Since the DEC instruction takes one word (two bytes), $–2 represents the first byte in the DEC instruction.

Caution should be exercised when using the $ symbol, since program logic errors could result. In the preceding example, an error might occur if an instruction were inserted between the DEC and JNE instructions without changing the $–2 expression. Inserting an instruction in the first example requires no other changes.

The symbols for the 9900 registers, register pairs, and memory address registers have been pre-defined by the assembler. However, their numeric values and additionally any data constant, I/O device address, or memory address in the operand field may be represented by expressions. An expression may consist of the following:

1) a single number,
2) a string constant,
3) a symbol, or
4) multiple numbers, string constants, and/or symbols combined with arithmetic and/or logical operations.

The assembler evaluates an expression in the operand field of a statement. If the expression violates permissible limits for the operand field, an error code is displayed. Additional information concerning expressions appears later in this section.

Any symbol appearing in the operand field other than the pre-defined symbols RØ through R15 and the location counter contents symbol, $, must be defined in the label field of a directive or any 9900 instruction in the source module, or in the operand field of a GLOBAL, STRING, SECTION, COMMON, or RESERVE directive.

A statement may contain both the operand symbol and its label definition, as in the case of an instruction that jumps to itself. For example:

LABEL	OPERATION	OPERAND	COMMENT
HERE	JEQ	HERE	;HANG HERE IF PREVIOUS RESULT ;IS NON-ZERO

Typically, however, the symbol is defined in another statement. If the symbol is not defined in any statement, an error code is displayed. Additionally, symbols appearing in the operand field of SET, EQU, ORG, and BLOCK directives must have been defined in the label field of a previous statement. Operand symbols in all other statements may be defined in the label fields of later statements.

If an illegal item appears in the operand field, the assembler flags the item with an error code on the display device and in the listing. All operand items are processed by the assembler in a 16-bit register. The assembler ignores any overflow conditions that occur while evaluating expressions. If the operand value is outside the range −32768 to 32767, an unflagged error in the object module may occur. If the operand requires a value within a smaller range and the value represented in the operand field is outside this range, a truncation error code is displayed and the appropriate number of least significant bits for the value is placed in the object module. Any undefined value in the operand field is treated as zero and an error code is displayed.

THE COMMENT FIELD

Programs containing comments are more readable, and hence easier to debug and modify. The optional comment field begins with a semicolon, is terminated by a carriage return, and follows all other statement fields. If no other fields are used, the comment field may begin anywhere in the statement.

String and macro substitution may be performed in the comment field. (Refer to the Section 2 subsection entitled String Text Substitution and to Section 5 for discussion on string and macro substitution.) Since the single quote character signals substitution, the character must be preceded by a caret (^) or up-arrow (↑) character when used for purposes other than substitution.

USING SYMBOLS

Symbol usage makes a program easier to read and modify, and reduces the risk of error during program modification. Symbols are defined when they appear in the label field of 9900 instructions, macro calls, and assembler directives, or in the operand field of GLOBAL, SECTION, COMMON, RESERVE, MACRO, or STRING directives. After having been defined, symbols can be used in the operation and operand fields of 9900 instructions, macro calls, and assembler directives.

A symbol label in a 9900 instruction represents the address of the first byte of that instruction. Such a label allows the user to transfer control (jump or branch) to an instruction without knowing its absolute address. To transfer control, place the instruction symbol in the operand field of the jump or branch instruction.

The meaning of a label symbol used as an operand for an assembler directive is dependent upon the directive. Generally, the symbol represents the memory address of data or a data constant value. Through the use of symbols, the directive operand field can refer to a data constant or a memory data area without regard to the absolute memory address. This is especially helpful when modifying a data constant frequently referred to by other statements. The programmer need only change the defining statement, rather than all statements referencing the constant.

Some symbols are created by the programmer, and others are pre-defined by the assembler.

Programmer-Defined Symbols

Programmer-defined symbols are assigned values during the assembler's first pass. Operand fields referring to the symbols are translated during the assembler's second pass. The ORG and BLOCK directives each alter the contents of the assembler location counter during both assembler passes. Because the alteration value is specified in the operand field of the ORG and BLOCK directives, any symbol appearing in the operand field of these directives must also be defined in the label field of a previous statement in the source module. The EQU directive operand field may contain a forward reference to a symbol, if the symbol does not appear in the operand field of an ORG, BLOCK, or another EQU directive. Forward referencing operand symbols are, however, allowed in all other statements. Symbols are permitted in the operand fields of EQU, SET, STRING, SECTION, COMMON, RESERVE, RESUME, GLOBAL, and MACRO directives.

Redefinition of symbols is generally not allowed. A previously defined SET symbol, however, may be redefined in another SET directive.

Pre-defined Symbols

Certain words are reserved as pre-defined symbol names for use in the operation and operand fields of source programs. Among these words are the following register symbols, assembler directives, instruction mnemonics, assembler listing options and operators. Refer to Appendix G for a complete list of reserved words for the 9900 Assembler.

FIELD	TYPE OF VALUE	PRE-DEFINED SYMBOLS
Operation	9900 Instruction mnemonics	Refer to Appendix C
Operation	Assembler directive mnemonics	Refer to Appendix B
Operand	Workspace registers	RØ through R15

Rules for Creating Symbols

The first character in a symbol must be alphabetic. The remainder of the symbol may be composed of the following characters: the letters A through Z; the numbers Ø through 9; and the special characters, . (period), _ (underscore), and $ (dollar sign). Lower-case letters are interpreted in their upper-case form. A symbol may contain up to eight characters. Only the first eight characters of the symbol are used, and excess characters are ignored. All pre-defined symbols are reserved words and cannot be redefined.

NUMERIC VALUES

The assembler defines two types of numeric values, scalars and addresses. Scalar values represent arbitrary numeric values. Address values represent actual memory locations within a program.

Scalar Values

Scalar values are signed integers ranging from –32,768 to +32,767. Scalar values serve as counting values in a program, rather than as actual references to memory locations. Scalar values are completely defined upon assembly.

Address Values

Address values represent actual memory locations within a user program. Address values are unsigned numbers ranging from Ø to 65,535. The assembler produces relocatable object code, that is, object code whose locations are defined during linking (see Section 9). Upon assembly, address values are relative to an assembler-defined base (or starting point). Therefore, actual memory locations associated with address values are unknown until after the linking process occurs.

More than one address base may exist within a given assembly. The user may define additional address bases by issuing a SECTION, COMMON, or RESERVE directive. Refer to Section 4 describing these directives and their relocation options. Since an address value lacks complete definition upon assembly, address value usage is more restrictive than scalar value usage. A unique location counter exists for each assembler-defined base in a user program. The $ symbol (current location counter contents) represents an address value.

NOTATION RULES FOR SPECIFYING CONSTANTS

Constants may be either numeric or string constants.

Numeric Constants

Numbers are integers and are assumed to be decimal unless otherwise specified. This means a number without a suffix is evaluated according to the decimal number base. A suffix letter code must be used to specify a radix other than decimal. The following suffixes are available:

1) H for hexadecimal. For example: 35H
   All numbers must begin with a numeric digit; therefore, a zero must precede all hexadecimal numbers beginning with the hexadecimal digits A through F. Examples of this follow:

   ØB5H and ØFFH

2) O (capital o, not zero) or Q for octal. For example: 76O and 76Q

3) B for binary. For example: 1Ø11Ø11ØB

Leading zeros are appended to or truncated from constants to produce 8- or 16-bit values as required by the particular operand. Blanks are not permitted within a numeric constant. Refer to Appendix E for hexadecimal, decimal, and binary number conversion tables.

String Constants

In addition to symbols and numeric constants, operations may also contain string constants. String constants can be generated by using ASCII strings. ASCII (American Standard Code for Information Interchange) is a standard code for representing characters transmitted between the computer and peripheral devices such as teletypes, printers, and terminals. String constants and variables may be combined into string expressions using special operators. A string expression may be used anywhere a normal expression is allowed. String constants are written by enclosing ASCII characters within double quotes ("). A string constant may contain any character within the source code character set except a carriage return.

A double quote character may be included within a string by preceding it with a caret character (^). The caret character removes the special meaning from any character and allows the special character to be treated as a regular part of the text. A caret may also be included within a string by entering two carets. Examples of string constants and caret usage follow:

"ABCDEF"	results in the string	ABCDEF
"123^"34"	results in the string	123"34
"^^"	results in the string	^

Null String

A string containing zero characters is a null string. A null string is entered as two double quotes without intervening text (" ").

String To Numeric Conversion

If a string expression is used where a numeric value is required, the string is automatically converted to a numeric value. The numeric value of a string is defined as follows:

The numeric value of the null string (" ") is zero.

The numeric value of a one-character string is a 16-bit value whose high order nine bits are zeros and whose low order seven bits contain the ASCII code for the character.

The numeric value of a two-character string is a 16-bit value as well. In this case, the ASCII code for the leftmost character is in the high-order byte. The ASCII code for the second character from the left is in the low-order byte.

The numeric value of a string longer than two characters is the numeric value of the leftmost two characters in the string. An error code is displayed when this occurs.

Examples of string to numeric conversion follow. The numeric values for ASCII characters are found in Appendix E.

STRING	NUMERIC VALUE
" "	Ø
"A"	41H
"12"	3132H
"123"	3132H (truncation error occurs)

EXPRESSIONS PERMITTED IN THE OPERAND FIELD

The operand field may contain an expression consisting of one or more terms acted on by expression operators. A term is either a symbol, a numeric constant, a string constant, or an expression enclosed within parentheses. The value of a term may be an address, a scalar value, or undefined. An address is an ordered pair, the first member being the base, the second member, the offset. The offset is known at assembly time, the base is not. A scalar value is represented by any integer. Spaces are permitted within an expression; the assembler reduces the expression to a single value. When an invalid term is used, the display device and the listing show an error code, and the value of the expression is undefined.

The following outline lists the expression operators and functions. A chart describing the hierarchy of all expression operators and functions follows this summary. Each expression operator and function is described in greater detail, completing this discussion.

Unary Arithmetic Operators

OPERATOR	MEANING
+	identity
–	sign inversion

Binary Arithmetic Operators

OPERATOR	MEANING
*	multiplication
/	division
+	addition
–	subtraction
MOD	remainder
SHL	shift left
SHR	shift right

Unary Logical Operator

OPERATOR	MEANING
\	not (bit inversion)

Relational Operators

OPERATOR	MEANING
=	equal
<>	not equal
>	greater than
>=	greater than or equal
<=	less than or equal
<	less than

Binary Logical Operators

OPERATOR	MEANING
&	and
!	inclusive or
!!	exclusive or

String Concatenation Operators

OPERATOR	MEANING
:	string concatenation

Functions

HI (exp)

Returns the most significant byte of a numeric expression. The expression may be either an address or a scalar value. If an address is specified as the HI function argument, subsequent operations must not be performed on the HI function result. The HI function result is numeric.

LO (exp)

Returns the least significant byte of a numeric expression. The expression may be either an address or a scalar value. If an address is specified as the LO function argument, subsequent operations must not be performed on the LO function result. The LO function result is numeric.

DEF (sym)

Returns −1 (true) if the symbol has been previously defined in this pass. Otherwise, returns Ø (false). The DEF function result is numeric.

SEG (string expression,exp1,exp2)

Extracts exp2 characters from the specified string, starting with the character, exp1. If the end of the string is encountered before exp2 characters are extracted, only those characters up to the string end are extracted. Both exp1 and exp2 must be scalar values. The SEG function result is a string.

NCHR (string expression)

Returns the current number of characters in the specified string. For a string variable, the length returned may be less than the length defined by the STRING directive. The NCHR function result is numeric.

ENDOF (section name)

Upon linking, the ENDOF function returns the address of the last byte of the specified section. The symbol specified in this function must be a global symbol. If the symbol is not a section name, the address of the symbol is returned. Further operations may be performed on the result of ENDOF, provided the operations are allowed for address values. The ENDOF function result is numeric.

BASE (exp1,exp2)

Returns −1 (true) if the two expressions, exp1 and exp2, share the same base. Otherwise, returns Ø (false). The BASE function result is numeric.

STRING (exp)

Returns the value of the expression as a six-character string. The five rightmost characters represent the decimal value of the expression; the leftmost character indicates whether the number is positive or negative. If the leftmost character is a minus, "−", the number is negative. If that character is a zero, "Ø", the number is positive. The expression must be a scalar value.

SCALAR (exp)

Converts the address value of the expression to a scalar value.

Hierarchy of Expression Operators and Functions

In multiple-operator expressions, operators and functions are evaluated in the order of their precedence. Table 2-1 illustrates this hierarchy. The functions at the top of the table have the highest precedence. The operators at the bottom of the table have the lowest precedence. All expression operators and functions located on the same line have equal precedence, and are evaluated from left to right. Parentheses may be used to override the order of precedence, and parentheses are evaluated from inward to outward. The most deeply parenthesized subexpressions are evaluated first.

If the expression entered is too complex for the assembler to translate, an expression error code is displayed. This does not occur when parentheses nesting depth is three or less.

LO	HI	SEG	NCHR	DEF	ENDOF	BASE	STRING	SCALAR
:								
+	– (unary plus and minus)			\				
*	/	SHL	SHR	MOD				
+	– (addition and subtraction)							
=	<>	<	<=	>	>=			
&								
!	!!							

Table 2-1. Hierarchy of Expression Operators and Functions.

Description of Expression Operators and Functions

In addition to the arithmetic (+, –, *, /) and logical (\, &, !, !!) operators, several other operators and functions are allowed within numeric expressions. These operators and functions provide additional arithmetic functions and a means for comparing numeric quantities.

Binary Arithmetic Operators

Binary arithmetic operators act on numeric values, which may be scalar or address values. Scalar values may appear within arithmetic operations in any combination. Only the following binary arithmetic operations are permitted when acting upon addresses:

SCALAR VALUE + ADDRESS = ADDRESS
ADDRESS + SCALAR VALUE = ADDRESS
ADDRESS – SCALAR VALUE = ADDRESS
ADDRESS – ADDRESS = SCALAR VALUE (Both addresses must be based to the same section.)

Any other combination of address terms yields an undefined result.

MOD is a binary operator that computes the remainder when the first operand is divided by the second operand. For example, an instruction entered as A MOD B yields the remainder resulting from A/B. The program segment that follows demonstrates MOD operator usage.

```
LABEL   OPERATION   OPERAND          COMMENT
AX      EQU         5 MOD 2          ;AX IS SET TO 1, SINCE 5/2 YIELDS A
                                     ;REMAINDER OF 1
BX      EQU         14 MOD AX        ;BX IS SET TO Ø, SINCE 14/1 YIELDS A
                                     ;REMAINDER OF Ø
CX      EQU         (BX + 29)MOD 25  ;CX IS SET TO 4, SINCE Ø+29 YIELDS 29
                                     ;AND 29/25 YIELDS A REMAINDER OF 4
DX      EQU         (–5) MOD 2       ;DX IS SET TO –1, SINCE –5/2 YIELDS A
                                     ;REMAINDER OF –1
```

SHL and SHR are binary operators that shift their first operands the number of bit positions specified by their second operands.

SHL performs a left logical shift (equivalent to multiplying by two). Zeros are shifted into the right end of the 16-bit value. Bits shifted out of the leftmost bit position are lost.

SHR performs a right logical shift. Zeros are shifted into the leftmost bit positions. Bits shifted from the rightmost bit position are lost. Shifts of 16 or more bits generate a result of zero and produce a truncation error code. The program segment that follows demonstrates SHL and SHR operator usage.

```
LABEL   OPERATION   OPERAND          COMMENT
DX      EQU         1 SHL 1          ;VALUE ASSIGNED TO DX IS 2, SINCE A
                                     ;SHIFT LEFT ONCE CAUSES 1 TO BE
                                     ;MULTIPLIED BY 2
EX      EQU         DX SHR 1         ;VALUE ASSIGNED TO EX IS 1 SINCE DX
                                     ;(2) SHIFTED RIGHT IN A BINARY FASHION
                                     ;YIELDS 1
FX      EQU         Ø6EØH SHL 3      ;VALUE ASSIGNED TO FX IS 37ØØH,
                                     ;SINCE 2 CUBED IS 8, AND 8 TIMES
                                     ;Ø6EØH IS 37ØØH
GX      EQU         ØFFFFH SHR 16    ;VALUE ASSIGNED TO GX IS Ø, SINCE
                                     ;ØFFFFH SHIFTED RIGHT IN A BINARY
                                     ;FASHION YIELDS Ø
```

Unary Operators

All unary operators may act upon scalar values. The plus sign (+) is the only unary operator permitted to act upon addresses.

Relational Operators

The relational operators include =, < >, >, <, < =, and > =. Relational operators allow signed numeric, unsigned numeric, and string comparisons.

Numeric Comparisons

If either of the operands of a relational operator is numeric, the relational operators perform signed or unsigned numeric comparisons. A signed numeric comparison is performed on two scalar values, a string and a scalar value, or a scalar and a string value. An unsigned numeric comparison is performed whenever one of the operands is an address. Comparison of two addresses based in different sections results in an error. These comparisons are summarized as follows:

	STRING	SCALAR	ADDRESS
STRING	String Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
SCALAR	Signed Numeric Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
ADDRESS	Unsigned Numeric Comparison	Unsigned Numeric Comparison	Unsigned Numeric Comparison

If a comparison is performed between an address and a string or scalar value, the base of the address is first added to the string or scalar value. If two addresses are compared, they must have the same base, or an error results.

For signed comparisons, numbers range from −32768 to 32767. For unsigned comparisons, numbers range from Ø to ØFFFFH (65,535).

An operator in a numeric comparison determines whether the specified relationship exists between its two operands. The resulting value is Ø if the relationship is false and −1 (ØFFFFH) if the relationship is true. Examples of relational operator usage follow.

```
LABEL   OPERATION   OPERAND     COMMENT
T       EQU         -5 > 7      ;VALUE ASSIGNED TO T IS Ø, SINCE -5
                                ;IS NOT GREATER THAN 7
P       EQU         7 > = -5    ;VALUE ASSIGNED TO P IS -1, SINCE 7
                                ;IS GREATER THAN -5
U       EQU         T < > P     ;VALUE ASSIGNED TO U IS -1, SINCE T
                                ;IS NOT EQUAL TO P
```

String Comparisons

The relational operators (=, < >, >, <, < =, > =) may be used to compare the values of two string expressions. When strings are compared using these relational operators, the comparison is made numerically, according to the ASCII collating sequence. Refer to Appendix E for the correct character ordering sequence of ASCII characters.

String comparison is performed only when both operands of a relational operator are strings. If only one of the operands of a relational operator is a string, the string is converted to a scalar value and a numeric comparison is performed.

String comparison always proceeds from left to right. If two strings are equal through the last character of the shorter string, the shorter string is considered to be less than the longer string.

Examples of string comparisons follow.

"AB" = "AB"	results in	−1 (true)
"AB" <> "AB"	results in	Ø (false)
"A" > "B"	results in	Ø, since A is less than B
"ABC" > "AAAA"	results in	−1, since B is greater than A
"ABC" > "ABC "	results in	Ø, since "ABC" has three characters and "ABC " has four, including the final space
"" < " "	results in	−1, since a null string is less than a blank character
1 < "1"	results in	−1, since the numeric value of the ASCII character "1" is 31H and is greater than 1

String Concatenation

The concatenation operation combines two strings into a single string. The operator used to specify string concatenation is the colon (:). The colon may be used to concatenate any two string expressions. An error occurs when an attempt is made to concatenate two numeric values or a string and a numeric value. Examples of string concatenation follow:

"A":"B"	results in	"AB"
"":""	results in	"", since two null strings produce a null string
"A":"":"B"	results in	"AB", since a null string and a character produce the character
"A":" "	results in	"A "
"ABC":"1":"2"	results in	"ABC12"

Functions

HI and LO are unary functions that respectively extract the high- and low-order eight bits of their operands. References to HI or LO are written as single argument functions. The value to be acted on appears in parentheses, following the keyword HI or LO. If this value is an address, further operations on the result of HI or LO are disallowed. Examples of HI and LO function usage follow:

```
LABEL   OPERATION   OPERAND              COMMENT
IXB     EQU         HI (ØCØØFH)          ;VALUE ASSIGNED TO IXB IS CØH
JX      EQU         LO (ØCØØFH)          ;VALUE ASSIGNED TO JX IS ØFH
KX      EQU         LO (HI(ØCØØFH) + 1)  ;VALUE ASSIGNED TO KX IS C1H
Z       EQU         5 + LO(Q)            ;INVALID WHEN Q IS AN ADDRESS
```

DEF is a unary function that determines whether a symbol has already been defined. DEF is referenced as a single-argument function. The argument must be a symbol and may not be an expression. If the argument symbol has already been defined, the value of DEF is −1 (ØFFFFH). If the argument has not been defined, the value of DEF is Ø. A pre-defined symbol used as an argument causes an error. Examples of DEF function usage follow.

```
LABEL   OPERATION   OPERAND     COMMENT
MK      EQU         DEF(K)      ;VALUE ASSIGNED TO MK IS −1 IF K IS
                                ;ALREADY DEFINED
Q       EQU         DEF(N)      ;VALUE ASSIGNED TO Q IS Ø IF N IS
                                ;UNDEFINED
RX      EQU         DEF(RX)     ;VALUE ASSIGNED IS Ø. THE SYMBOL ON
                                ;THE LEFT OF THE EQU DIRECTIVE IS
                                ;UNDEFINED UNTIL THE EXPRESSION
                                ;ON THE RIGHT IS EVALUATED
S       WORD        DEF(S)      ;A WORD OF OBJECT CODE CONTAINING
                                ;ØFFFFH(−1) IS GENERATED. THE LABEL
                                ;ON THE WORD STATEMENT IS DEFINED
                                ;BEFORE THE STATEMENT IS EVALUATED
```

The SEG function (segmentation) is used to extract a portion of a string. The SEG function uses three arguments. The first argument is the string (or string expression) from which a substring is to be extracted. The second argument is a numeric expression specifying the position of the leftmost character of the string where the substring is to be extracted. Characters within the string are numbered from left to right starting with one. The third argument is a numeric expression specifying the number of characters to be extracted. The specified characters are extracted unless the end of the string is encountered first. In this case, only those characters up to the end of the string are extracted. The following examples illustrate properties of the SEG function:

SEG("ABCD",2,2)	results in	"BC"
SEG("ABCD",1,4)	results in	"ABCD"
SEG("ABCD",3,3)	results in	"CD"
SEG("ABCD",5,2)	results in	""(the null string, resulting in zero characters)
SEG("ABCD",3,Ø)	results in	""

The NCHR function may be used to determine the number of characters in a string expression. NCHR is referenced as a single-argument function, that argument being the string expression whose length is to be determined. The result of NCHR is numeric and not a string value. Examples of NCHR function results follow.

NCHR ("")	results in	Ø
NCHR("ABC")	results in	3
NCHR(SEG("XYZ",2,1)	results in	1
SEG("ABC",NCHR("ABC") ,1)	results in	"C", since C is the last character of "ABC"

The ENDOF function returns the address of the last byte of a section. The argument for ENDOF must be a global symbol whose ending address is to be determined. If the global symbol is not the name of a section, the result is the address of the symbol. An example of ENDOF usage follows:

```
LABEL     OPERATION     OPERANDS          COMMENT
          .
          .
          .
          RESERVE       STACK,100H        ;NAMES A SECTION, STACK, AND
                                          ;ALLOCATES AT LEAST 256 BYTES
          LI            R1,ENDOF(STACK)   ;LOAD REGISTER R1 WITH THE END
                                          ;OF THE STACK
```

The BASE function determines whether two expressions share the same base. If the expressions share the same base, the value of BASE is true (ØFFFFH). Otherwise, the value of BASE is false (Ø). Examples of BASE function results follow. Q,R, and ZZ represent addresses where Q and R share a common base, while ZZ does not.

BASE (Q,R)	results in	ØFFFFH (true)
BASE (Q,Q+15)	results in	ØFFFFH (true)
BASE (ZZ,Q)	results in	Ø (false)
BASE (Q,Q−R)	results in	Ø (false) because Q−R is scalar
BASE (5,15)	results in	ØFFFFH (true) because 5 and 15 are both scalar
BASE (5,Q−R)	results in	ØFFFFH (true)
BASE (5,ZZ−Q)	results in	Error since subtraction is not valid between addresses with different bases

The STRING function returns the decimal value of an expression as a six-character string. The expression must be a scalar value. When the value does not fill six digits, leading zeros appear in the resulting string. If the expression value is negative, a minus sign is placed in the resulting string. Examples of STRING function results follow:

STRING(5)	results in	"ØØØØØ5"
STRING(5+15)	results in	"ØØØØ2Ø"
STRING(ØFFH)	results in	"ØØ255"
STRING(−ØFFH)	results in	"−Ø0255"

The SCALAR function converts the address value of the expression to a scalar value. The resulting scalar value is equal to the displacement of the address value from the address value's base. Upon linking, the resulting scalar value might not be the same as the final value of the expression. The SCALAR function does not affect scalar-valued expressions.

An example of scalar conversion follows:

```
LABEL   OPERATION   OPERAND                COMMENT
        SECTION     X                      ;DEFINES A NEW SECTION NAMED
                                           ; X
A1      ORG         7                      ;ADVANCES LOCATION COUNTER
                                           ;TO ADDRESS 7. ASSIGNS ADDRESS
                                           ;7 TO A1
        WORD        SCALAR($)MOD2          ;CONVERTS ADDRESS 7 TO SCALAR
                                           ;VALUE. PERFORMS 7/2 AND
                                           ;RETAINS REMAINDER 1
                                           ;ALLOCATES ONE WORD TO
                                           ;VALUE 1
        SECTION     ASDF                   ;DEFINES NEW SECTION NAMED
                                           ;ASDF
A2      ORG         6                      ;ADVANCES LOCATION
                                           ;COUNTER TO ADDRESS 6 WITHIN
                                           ;SECTION ASDF. ASSIGNS 6 TO A2
        WORD        SCALAR(A1)+SCALAR(A2)  ;ALLOCATES ONE WORD
                                           ;CONTAINING SCALAR VALUE 13
```

Note that if the SCALAR function were not entered in the above WORD directives, an error would result. Scalar values are unaffected by changes in address base. Thus, in the above program, the scalar result of the operation WORD SCALAR(A1)+SCALAR(A2) remains unchanged no matter what base values are assigned to sections X and ASDF upon linking.

STRING VARIABLES

String variables enhance the value of string expressions by providing a means for storing string expression values. A string variable is a symbol with an associated string value, and is created by use of the STRING directive.

The desired string variable names are defined in the operand field of the STRING statement. The maximum character length of the value to be stored in the string variable may be specified by entering a numeric expression in the operand field. When this optional character length expression is not specified, an eight-character length is assumed. In the following example, a string variable is defined as STRVAR, with a maximum character length of 16.

```
LABEL     OPERATION     OPERAND
          STRING        STRVAR(16)
```

For further discussion pertaining to STRING statements, refer to Section 4 describing assembler directives.

SET Strings

The SET directive assigns a string expression value to a string variable defined with the STRING directive. The string variable is entered in the label field of the SET directive; the string expression is entered in the operand field. The string expression value is evaluated and assigned to the string variable. If the resulting string expression's length is longer than the maximum string variable length, the string expression is truncated before assignment, and an error code is displayed. Examples of SET string usage follow.

```
LABEL     OPERATION     OPERAND              COMMENT
          STRING        A1,A2(2),A3(45),A4(Ø)  ;DEFINES STRING VARIABLE A1
                                               ;WITH A DEFAULTING VALUE
                                               ;LIMIT OF 8 CHARACTERS
                                               ;DEFINES STRING VARIABLES
                                               ;A2, A3, AND A4 WITH
                                               ;RESPECTIVE VALUE LIMITS OF
                                               ;2, 45, AND Ø CHARACTERS
A1        SET           "AB"                 ;VALUE OF A1 IS "AB"
A2        SET           A1                   ;VALUE OF A2 IS "AB"
A4        SET           A1:A2                ;VALUE OF A4 IS ""
                                               ;TRUNCATION ERROR SINCE A4
                                               ;ALLOWS A VALUE LIMIT OF Ø
                                               ;CHARACTERS
```

(This program continued on next page)

```
LABEL   OPERATION   OPERAND                 COMMENT
A3      SET         "A MEDIUM LONG STRING"  ;VALUE OF A3 IS "A
                                            ;MEDIUM LONG STRING"
A1      SET         A3                      ;VALUE OF A1 IS "A MEDIUM",
                                            ;STRING TRUNCATED
```

String Text Substitution

String variables may be used for modification of source text being processed by the assembler. Using string variables makes it possible to insert code into a source line, thus allowing the code to be processed as if it were part of the original source line. Before the assembler processes a source line, it scans the line for string variables enclosed within single quote characters. When such a variable is encountered, it is replaced with the specified value and the scan continues. When the entire line has been scanned and all code substitutions are made, the assembler then processes the line. For example assume the assembler processes the following code.

```
LABEL   OPERATION   OPERAND
        STRING      OP
OP      SET         "WORD"
        'OP'        1,2,3
```

When the assembler scans the line containing 'OP' 1,2,3, the string variable 'OP' is replaced with the value defined for the substitution, "WORD". The line resulting upon assembly follows:

```
        WORD 1,2,3
```

String substitutions can occur anywhere within a line of code, including within string constants and comments. For the examples that follow, assume that A1, A2, A3, and A4 are defined as specified.

LABEL	OPERATION	OPERAND
	STRING	A1,A2,A3,A4
A1	SET	"YTE"
A2	SET	"123,456"
A3	SET	"COMMENT"
A4	SET	""

Assume that the following substitutions are then performed.

SOURCE CODE	RESULTS AFTER SUBSTITUTION
BYTE 'A1','A2'	BYTE YTE,123,456
WORD 1 'A4'	WORD 1
A4 SET " 'A3' "	A4 SET "COMMENT"
WORD " 'A4' "	WORD "COMMENT"
B'A1' 'A2'–200	BYTE 123,456–200
B'A1"A2'	BYTE123,456 (error code displayed due to undefined instruction mnemonic, since space was omitted 'A1' and 'A2')

Since the single quote character always signals string substitution, it is necessary to precede the character with a caret (∧) if string replacement is not to be performed. The caret character allows the single quote character to then be interpreted as a literal character in a statement. An example demonstrating caret usage follows.

ASCII "WHAT∧'S UP DOC?" results in WHAT'S UP DOC?

Section 3

STATEMENT SYNTAX CONVENTIONS

INTRODUCTION

Many of the following sections in this manual contain TEKTRONIX Assembler and TEKDOS statement descriptions. Each statement description is preceded by a syntactical block showing the required statement format. This section describes the syntax conventions for TEKTRONIX Assembler and TEKDOS statements.

TEKTRONIX ASSEMBLER STATEMENT SYNTAX

TEKTRONIX Assembler directives and macro calls may contain up to four fields. Each field name is indicated in the syntactical block above the corresponding field item, as shown in the following example.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BYTE	{expression} [,expression]	[;charstring]

Use of Upper and Lower Case Letters and Punctuation

A capitalized item in a field must be entered exactly as shown. Punctuation delimiters such as commas, semicolons, or parentheses must also be entered exactly as shown. Spaces or tab characters terminate each field and begin the next. An item shown in lower case letters is a term signifying the entry type. The following descriptive terms are used to signify entry type unless otherwise specified:

1) symbol – as defined in Section 2
2) expression – as defined in Section 2
3) charstring – a string of one or more characters.

Blank Fields

Any field left blank is an illegal field for that statement.

Braces and Brackets

When an item is enclosed in braces { } , the item must be present in the statement. Items enclosed in brackets [ ] are optional. Braces and brackets are used for syntactical representation only and should not be entered as part of the statement. Braces and brackets may be nested. The following is an example of braces and brackets nested in braces.

$$\left\{\{\text{strvar1}\}\quad[\text{lenexp1}]\right\}$$

Trailing Dots

A line of dots following an item indicates that the item can be repeated a number of times. The item cannot be repeated beyond the end of the line being entered. In the example that follows, the item can be repeated.

[,symbol]...

TEKDOS STATEMENT SYNTAX

A TEKDOS statement contains a command and in most cases, one or more parameters with delimiting characters. An example of a typical TEKDOS statement syntactical block follows.

SYNTAX

NUDGE {filename} [device / filename [/disc drive]] [{line number 1} {line number 2}]...

Command Name

A minimum set of characters (short form) is required for each TEKDOS command. This minimum character set is underlined in the syntactical description. In the page heading for the command, the exact spelling of the command name is given with the short form underlined. Commands without a short form are not underlined.

In addition to the minimum set of characters in the command name, a maximum set (long form) is also given for each command name. Any number of characters in the command name, ranging from the short form spelling to the long form spelling, may be used as long as the exact spelling is followed.

Delimiters

Items in the command line must be separated by delimiters when entered into the terminal. A space is used as the main delimiter. The slash (/) is used to delimit a file name and the disc drive number.

The comma may be used as a delimiter in most cases. Two commas are used to specify null or empty fields in a parameter list. Three commas are used to specify two adjacent null fields.

Parameters

The parameters or controlling conditions of each command line are shown in the preceding TEKDOS statement syntactical block. These parameters may be names, numbers, characters or symbols. When a parameter is shown capitalized, it must be entered exactly as shown. Parameters shown in lower case letters are descriptive terms to signify the type of entry.

Braces and Brackets

When appearing in TEKDOS statement syntactical descriptions, braces and brackets have the same meaning as when used with TEKTRONIX Assembler statements. Additionally, parameters stacked within either braces or brackets indicate that only one of the enclosed items should be selected for statement entry. In the following example, an object file name or an object device may be selected, but not both.

[object filename
  object device]

Trailing Dots

Trailing dots within TEKDOS statement syntactical blocks indicate repetitive parameters.

Section 4

ASSEMBLER DIRECTIVES

INTRODUCTION

The following assembler directives are available:

Listing Format Control Directives

- LIST
- NOLIST
- PAGE
- SPACE
- TITLE
- STITLE
- WARNING

Symbol Definition Directives

- EQU
- STRING
- SET

Workspace Location Determination Directive

- WPNT

Location Counter Control Directive

- ORG

Data Storage Control Directives

- BYTE
- WORD
- ASCII
- BLOCK

Macro Definition Directives

MACRO
ENDM
REPEAT
ENDR
INCLUDE

Conditional Assembly Directives

IF
ELSE
ENDIF
EXITM

Relocatable Section Definition Directives

SECTION
COMMON
RESERVE
RESUME
GLOBAL
NAME

Module Termination Directive

END

LISTING FORMAT CONTROL DIRECTIVES

The assembler listing format directives are presented in the order shown below:

Mnemonic	Purpose
LIST	Enables display of assembler listing features.
NOLIST	Disables display of assembler listing features.
PAGE	Begins the next listing line on the following page.
SPACE	Spaces downward a specified number of listing lines.
TITLE	Creates a text line at the top of each listing page for program identification.
STITLE	Creates a text line on the second line of each listing page heading for program identification.
WARNING	Upon assembly, generates a warning message on the output device and in the listing. Also allows the user to specify his own warning message.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	LIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]

PURPOSE

Two assembler listing control directives, LIST and NOLIST, respectively enable and disable display of assembler listing features.

EXPLANATION

When NOLIST is specified without operands, all output to the listing file (except the symbol table) is suppressed. When LIST is entered without operands, the listing is turned back on.

General Listing Format Control Options

Four general listing control options (CND, TRM, SYM, and CON) may be entered with the listing control directive, LIST, when specific features in the assembler listing are desired for viewing. The same four listing options may be entered with the assembler listing control directive, NOLIST, when specific features in the assembler listing are not desired for viewing.

The general listing control options are summarized as follows:

CND – Lists unsatisfied conditions for IF and REPEAT operations. (Refer to the subsections describing macro definition directives and conditional assembly directives.) The listing defaults to an OFF condition, thus displaying only those instructions within an IF or REPEAT condition occurring when the condition is satisfied.

TRM – Causes the listing to be trimmed to a 72-character format during display. Defaults to an OFF condition, causing the listing to be displayed in the standard 132-character format.

SYM – Lists the symbol table. Defaults to an ON condition.

CON – Displays all assembly errors to the console. Defaults to an ON condition.

Macro Listing Format Control Options

A macro is a shorthand approach for inserting a pre-defined source code block into a program. Refer to Section 5 for a discussion of macro procedures.

Only those macro instructions generating object code appear in an assembler listing. Some of the code generated during a macro expansion does not generate object code upon assembly, making it impossible under normal conditions to view the entire macro expansion sequence within the assembler listing. Therefore, in addition to the four general listing control options, two macro listing control options (MEG and ME) may be entered with the LIST and NOLIST directives to enable and disable macro expansion visibility. These options are summarized as follows:

MEG – Lists only macro expansion code that changes the location counter. Defaults to an ON condition.

ME – Lists all macro expansion code except for any unsatisfied IF or REPEAT conditions. When the listing control option CND is on, unsatisfied conditions are also listed. Defaults to an OFF condition. If either ME or MEG is turned OFF by the user, the other is automatically turned OFF. If ME is turned ON by the user, MEG is automatically turned ON.

The following table demonstrates LIST and NOLIST effects on the ME and MEG options:

ENTRY	RESULTS	
NOLIST MEG	MEG is OFF	ME is OFF
NOLIST ME	MEG is OFF	ME is OFF
LIST MEG	MEG is ON	ME is OFF
LIST ME	MEG is ON	ME is ON
NOLIST	MEG is OFF	ME is OFF
	Status of both options is saved	
LIST	Restores status of both options	

Upon exit from a macro expansion, the main program listing status is restored to the status that prevailed before the macro was called.

Conventions for Listing Control

The LIST and NOLIST directives are always entered in the operation field of the listing control statements where they appear. More than one listing control option may be entered with the LIST and NOLIST directives. In this case, each option is separated from other options by a comma. When entering the listing control options with the LIST or NOLIST directives, the options are placed in the operand field of the listing control directive in any order. If the NOLIST directive is entered without options to suppress display, and the LIST directive is again entered without options specified, the original specified options are retained. The number on any listing line corresponds to the original input source line number. The NOLIST directive does not affect this line number correlation.

EXAMPLE

The following listing control statement suppresses the symbol table listing.

```
LABEL     OPERATION     OPERAND     COMMENT
          NOLIST        SYM         ;SUPPRESSES SYMBOL TABLE LISTING
```

The following listing control statement causes all subsequent macro expansions and unsatisfied conditions to be included within the assembler listing.

```
LABEL     OPERATION     OPERAND     COMMENT
          LIST          ME,CND      ;LISTS MACRO EXPANSIONS
                                    ;AND ALL UNSATISFIED
                                    ;CONDITIONS
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	PAGE		[;charstring]

PURPOSE

The PAGE directive causes the next listing line to begin on the following page.

EXPLANATION

As the source lines are read by the assembler in its second pass, they are output to the listing along with any object code produced. When the PAGE directive is encountered, a page heading is printed at the top of the new page and the next listing line begins below the heading. The actual PAGE directive is not printed in the listing.

A label is generally not used with the PAGE directive; however, if used, the symbol represents the address in the assembler location counter. The location counter contains the address of the next instruction or data byte in the program sequence.

EXAMPLE

The following program illustrates PAGE directive usage:

```
LABEL   OPERATION   OPERAND      COMMENT
        STRING      S1 (8Ø)      ;DEFINE STRING VARIABLE S1
                                 ;WITH 8Ø - CHARACTER
                                 ;MAXIMUM
L1      EQU         3            ;DEFINE CONSTANT SYMBOL
                                 ;L1 TO EQUAL 3
L2      SET         4            ;DEFINE VARIABLE SYMBOL L2
                                 ;TO EQUAL 4
        PAGE                     ;BEGINS NEW LISTING PAGE
        ORG         1100H        ;STARTS OBJECT CODE OF NEXT
                                 ;INSTRUCTION AT 1100H
        MOV         R1,R2        ;LOADS THE CONTENTS OF R1
                                 ;INTO R2
        END                      ;END OF PROGRAM
                                 ;REG. A
```

Upon assembly, the following listing file results from this source program. A new page is generated after the SET directive.

```
TEKTRONIX 9900 ASM Vx.x                                    PAGE 1
ØØØØ1                    STRING  S1 (8Ø)    ;DEFINE STRING VARIABLE S1
                                            ;WITH 8Ø -CHARACTER
                                            ;MAXIMUM
ØØØØ2   ØØØ3  L1         EQU     3          ;DEFINE CONSTANT SYMBOL
                                            ;L1 TO EQUAL 3
ØØØØ3   ØØØ4  L2         SET     4          ;DEFINE VARIABLE SYMBOL
                                            ;L2 TO EQUAL 4
.
.
.
```

(Program continued on next page)

```
TEKTRONIX 9900 ASM Vx.x                                          PAGE 2
00005        1100  >          ORG    1100H        ;STARTS OBJECT CODE OF
                                                   ;NEXT INSTRUCTION AT 1100H
00006  1100  C081             MOV    R1,R2        ;LOADS THE CONTENTS OF R1
                                                   ;INTO R2
00007                          END                 ;END OF PROGRAM
.
.
.
TEKTRONIX 9900 ASM Vx.x    SYMBOL TABLE LISTING                  PAGE 3

STRINGS AND MACROS

   S1 - - - - - 0050 S

SCALARS
R0   --- 0000    R1   --- 0001    R2   --- 0002
R3   --- 0003    R4   --- 0004    R5   --- 0005
R6   --- 0006    R7   --- 0007    R8   --- 0008
R9   --- 0009    R10  --- 000A    R11  --- 000B
R12  --- 000C    R13  --- 000D    R14  --- 000E
R15  --- 000F    L1   --- 1100    L2   --- 0004V
SI   --- 0050S

% (default) SECTION (0102)

7 SOURCE LINES   7 ASSEMBLED LINES   1000 BYTES AVAILABLE
```

Note that the symbol indicators V and S respectively follow the symbols L2 and S1. The symbol indicator V indicates that L2 is a SET symbol. The symbol indicator S indicates that S1 is a string. The symbol L1 has no symbol indicator following it, indicating that L1 is an EQU symbol. For a more complete description of symbol indicators, refer to Section 7, entitled ASSEMBLER LISTING FORMAT.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	SPACE	[expression]	[;charstring]

PURPOSE

Whenever the SPACE directive appears in the source module, the assembler spaces downward a specified number of lines in the listing.

EXPLANATION

The number of lines to be spaced downward is indicated by the expression in the SPACE directive operand field. If no expression is entered, one space is generated. If the execution of the SPACE directive crosses a page boundary, the effect is the same as that of the PAGE directive. The actual SPACE directive is not printed in the listing.

A label is generally not used with the SPACE directive; however, if used, the symbol represents the address in the assembler location counter. The location counter contains the address of the next instruction or data byte in the program sequence.

EXAMPLE

Assume the following source program resides on disc.

```
LABEL   OPERATION   OPERAND      COMMENT
        STRING      S1 (8Ø)      ;DEFINE STRING VARIABLE S1
                                 ;WITH 8Ø-CHARACTER MAXIMUM
L1      EQU         3            ;DEFINE CONSTANT SYMBOL
                                 ;L1 TO EQUAL 3
L2      SET         4            ;DEFINE VARIABLE SYMBOL
                                 ;L2 TO EQUAL 4
        SPACE       1Ø           ;SPACES DOWNWARD 1Ø
                                 ;LISTING LINES
        ORG         11ØØH        ;STARTS OBJECT CODE OF
                                 ;NEXT INSTRUCTION AT 11ØØH
        MOV         R1,R2        ;LOADS THE CONTENTS OF R1
                                 ;INTO R2
        END                      ;END OF PROGRAM
```

Upon assembly, the following listing file results from this source program. Ten lines are generated between the SET and ORG directives.

```
TEKTRONIX 9900 ASM Vx.x                                          PAGE 1
0000l                          STRING  S1 (80)      ;DEFINE STRING VARIABLE S1
                                                    ;WITH 80-CHARACTER
                                                    ;MAXIMUM
00002        0003  L1          EQU     3           ;DEFINE CONSTANT SYMBOL
                                                    ;L1 TO EQUAL 3
00003        0004  L2          SET     4           ;DEFINE VARIABLE SYMBOL
                                                    ;L2 TO EQUAL 4


00005        1100  >           ORG     1100H       ;STARTS OBJECT CODE OF
                                                    ;NEXT INSTRUCTION AT 1100H
00006 1100 C081                MOV  R1,R2          ;LOADS THE CONTENTS OF R1
                                                    ;INTO R2
00007                          END                 ;END OF PROGRAM

.
.
.
.
TEKTRONIX 9900 ASM Vx.x    SYMBOL TABLE LISTING                  PAGE 2

STRINGS AND MACROS

  S1 - - - - - 0050 S

SCALARS

  R0  --- 0000    R1  --- 0001    R2  --- 0002
  R3  --- 0003    R4  --- 0004    R5  --- 0005
  R6  --- 0006    R7  --- 0007    R8  --- 0008
  R9  --- 0009    R10 --- 000A    R11 --- 000B
  R12 --- 000C    R13 --- 000D    R14 --- 000E
  R15 --- 000F    L1  --- 1100    L2  --- 0004V
  SI  --- 0050S

% (default) SECTION (0102)

7 SOURCE LINES   7 ASSEMBLED LINES   1000 BYTES AVAILABLE
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	TITLE	{string expression}	[;charstring]

PURPOSE

The TITLE directive creates a text line at the top of each listing page for program identification.

EXPLANATION

The character string specified as the TITLE operand is printed in the page heading between the assembler version number and the page number. As many as 31 characters may be entered. Any characters exceeding the 31-character limit are truncated. The actual TITLE directive is not printed on the listing.

EXAMPLE

Assume the following TITLE statement is entered in a source program:

LABEL	OPERATION	OPERAND
	TITLE	"THIS IS THE PROGRAM TITLE"

Upon assembly, the specified title appears within the heading at the top of each listing page of the program as follows:

```
TEKTRONIX 9900 ASM Vx.x  THIS IS THE PROGRAM TITLE                PAGE 1
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	STITLE	{string expression}	[;charstring]

PURPOSE

The STITLE directive creates a text line on the second line of each listing page heading for program identification.

EXPLANATION

The character string specified as the STITLE operand is printed between the page heading and the first source code line. A blank line is automatically inserted between the string and the beginning of the source code. As many as 72 characters may be entered. Any characters exceeding the 72-character limit are truncated. The actual STITLE directive is not printed on the listing.

EXAMPLE

Assume the following STITLE statement is entered in a source program.

LABEL	OPERATION	OPERAND
	STITLE	"THIS LINE DEMONSTRATES STITLE USAGE"

Upon assembly, the specified STITLE line appears within the heading at the top of each listing page as follows:

```
TEKTRONIX 9900 ASM Vx.x                                    PAGE 1
THIS LINE DEMONSTRATES STITLE USAGE
(blank line)
.
.  (source code)
.
.
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	WARNING		[message]

PURPOSE

When an error is suspected within source code, the WARNING directive can be entered to generate an error message at assembly time. Thus, the nature of the errors in a program can be described upon assembly and listing.

EXPLANATION

A warning message may be entered as a comment in the WARNING directive. Unlike other comments, the warning message is not preceded by a semicolon. Upon assembly, this optional message is printed on the assembly listing and on the output device, flagging the suspected error. The following assembler message is also displayed on both the assembler listing and the output device during assembly, below the specified warning message:

```
***** ERROR 001
```

EXAMPLE

Assume the following WARNING directive is entered within a source program below a line containing an error.

```
LABEL     OPERATION          COMMENT
          WARNING            **** ENTRY OUT OF SEQUENCE
```

Upon assembly, the specified warning line appears below the source line containing the error. The message, ***** ERROR ØØ1 also appears below the specified warning message.

```
.
.
.
ØØØC ØØØ3 + LEN SET NCHR("ABB")
ØØØD       WARNING            ****ENTRY OUT OF SEQUENCE
***** ERROR ØØ1
.
.
.
```

SYMBOL DEFINITION DIRECTIVES

The assembler symbol definition directives are presented in the order shown in the following summary.

Mnemonic	Purpose
EQU	Permanently assigns a value to a symbolic name.
STRING	Declares the named statement symbols as string variables.
SET	Assigns or reassigns an expression's value to a string or numeric variable symbol.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
{symbol}	EQU	{expression}	[;charstring]

PURPOSE

The EQU directive permanently assigns a value to a symbolic name.

EXPLANATION

The symbol in the label field of an EQU directive is the symbolic name and the expression in the operand field represents the value. The symbol acquires the same base as the operand expression. No redefinition of this symbol is permitted.

The EQU directive operand field may contain a forward reference to a symbol label if the symbol does not appear in the operand field of an ORG, BLOCK, or another EQU directive.

If a symbol is declared in a GLOBAL directive and is defined by an EQU directive, the expression in the operand field of the EQU directive may not contain a HI, LO, or ENDOF function applied to an address. An error results when this occurs.

EXAMPLE

The following line demonstrates EQU directive usage:

```
LABEL   OPERATION   OPERAND   COMMENT
L1      EQU         3         ;ASSIGNS THE VALUE 3 TO THE
                              ;CONSTANT SYMBOL L1.
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	STRING	{{strvar1} [(lenexp1)]}[{,strvar2} [(lenexp2)]]...	[;charstring]

PURPOSE

The STRING directive declares the symbols named in the statement to be string variables.

EXPLANATION

The STRING directive declares the symbols "strvar1" and "strvar2" to be string variables. A string variable is a symbol with an associated string value. Numeric expressions "lenexp1" and "lenexp2" may be optionally entered next to the string variables to specify the maximum character length of the values stored in the string variables. This maximum character length must be a scalar value greater than or equal to zero. When the optional character length expression is not specified, an eight-character maximum length is assumed. If the optional character length expression is specified, it must be enclosed within parentheses. An operand symbol named in a statement containing the optional character length expression must not be a forward reference.

A symbol must be declared with the STRING directive before it may be used as a string variable. Symbols declared as string variables must not be used for any other purpose within a program. Any number of string variables may be declared with the STRING directive. When a string variable is initially declared, its value is the same as that of the null string.

EXAMPLES

The following examples demonstrate STRING directive usage:

```
LABEL   OPERATION   OPERAND                    COMMENTS
        STRING      STR(14)                    ;DECLARES STR
                                               ;AS A STRING
                                               ;VARIABLE WITH
                                               ;A MAXIMUM
                                               ;CHARACTER
                                               ;LENGTH OF 14
        STRING      A1,A2,A3,A4,X(NCHR("1234")) ;DECLARES A1
                                               ;THROUGH A4
                                               ;AS STRING
                                               ;VARIABLES
                                               ;WITH A
                                               ;MAXIMUM
                                               ;CHARACTER
                                               ;LENGTH OF 8.
                                               ;DECLARES X AS
                                               ;A 4-CHARACTER
                                               ;STRING
                                               ;VARIABLE SINCE
                                               ;THE NUMBER
                                               ;OF CHARACTERS
                                               ;IN "1234" IS 4.
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
{symbol}	SET	{expression}	[;charstring]

PURPOSE

The SET directive is used to assign or reassign an expression value to a string or numeric variable symbol.

EXPLANATION

The string or numeric variable symbol is entered in the label field of a SET directive. A string variable symbol must have first been defined with the STRING directive. A numeric variable symbol must not have been previously defined, unless by another SET directive. Variable symbols may not be subsequently redefined as labels, or be redefined by an EQU, STRING, SECTION, COMMON, RESERVE, GLOBAL, or MACRO directive. The value of a variable symbol may, however, be redefined by another SET directive.

The expression value is entered in the operand field. The expression is then evaluated and the value is assigned to the variable symbol.

If a SET directive contains a string-valued symbol and a numeric-valued expression, the numeric expression is converted to a string. This conversion is valid only when the numeric expression is a scalar value. The decimal value of the numeric expression is assigned to the string-valued symbol. The assigned string is six characters long, with the leftmost character being a minus sign if the value is negative. All numeric values are prefixed with leading zeros if less than six characters long. The numeric-expression to string-symbol conversion process is diagrammed as follows:

LABEL	OPERATION	OPERAND	COMMENT
string	SET	numeric	;RESULTS IN EXPRESSION ;CONVERSION TO STRING

If the SET directive contains a numeric-valued symbol and a string-valued expression, the string expression is converted to a numeric value. Refer to Section 2 of this manual, ASSEMBLER SOURCE MODULE FORMAT, which describes String to Numeric Conversion. The string-expression to numeric-symbol conversion process is diagrammed as follows:

LABEL	OPERATION	OPERAND	COMMENT
numeric	SET	string	;RESULTS IN EXPRESSION ;CONVERSION TO NUMERIC

Conversion is not required when a string-valued symbol is set to a string expression or a numeric-valued symbol is set to a numeric expression. When a symbol is set to an expression value, the symbol acquires the same section as the expression.

For string variable symbols where the length of the resulting expression value exceeds the maximum symbol string length, the expression value is truncated on the right before assignment. A truncation error code is then displayed.

EXAMPLES

Examples of typical SET instructions and the resulting string-valued symbol expression values follow:

LABEL	OPERATION	OPERAND	COMMENT
	STRING	A1,A2(2),A3(45),A4(Ø)	;DEFINES STRING VARIABLE ;A1 WITH A DEFAULTING ;VALUE LIMIT OF 8 ;CHARACTERS. DEFINES ;STRING VARIABLES A2, A3, ;AND A4 WITH RESPECTIVE ;VALUE LIMITS OF 2, 45, AND ;Ø CHARACTERS

(Program continued on next page)

```
LABEL   OPERATION   OPERAND                 COMMENT
A1      SET         "AB"                    ;VALUE OF A1 IS "AB"
A2      SET         A1                      ;VALUE OF A2 IS "AB"
A4      SET         A1:A2                   ;VALUE OF A4 IS "",
                                            ;TRUNCATION ERROR SINCE
                                            ;A4 ALLOWS A VALUE OF
                                            ;ONLY Ø CHARACTERS
A3      SET         "A MEDIUM LONG STRING"  ;VALUE OF A3 IS "A MEDIUM
                                            ;LONG STRING"
A1      SET         A3                      ;VALUE OF A1 IS "A MEDIUM",
                                            ;TRUNCATION ERROR
```

The following example demonstrates string-to-numeric and numeric-to-string expression conversion.

```
LABEL   OPERATION   OPERAND     COMMENT
        STRING      A1,A2       ;DEFINES STRING VARIABLES
                                ;A1 AND A2
A1      SET         14          ;VALUE OF A1 IS "ØØØØ14"
A2      SET         -1          ;VALUE OF A2 IS "-ØØØØ1"
A1      SET         5EH         ;VALUE OF A1 IS "ØØØØ94"
B1      SET         A2          ;NUMERIC SYMBOL, B1, IS SET
                                ;TO THE NUMERICALLY
                                ;CONVERTED EXPRESSION, A2.
                                ;TRUNCATION ERROR OCCURS,
                                ;SINCE A2 IS GREATER THAN
                                ;TWO CHARACTERS (-ØØØØ1).
                                ;THE TWO RESULTING
                                ;LEFTMOST ASCII CHARACTERS
                                ;ARE -Ø, GIVING B1 A
                                ;NUMERIC SET VALUE OF
                                ;2D3ØH
```

WORKSPACE LOCATION DETERMINATION DIRECTIVE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	WPNT	[/] expression	[;charstring]

PURPOSE

The WPNT directive tells the assembler the location of your current workspace.

EXPLANATION

The operand expression defines the location of the current workspace. Subsequent instructions containing symbolic addresses that are within 32 bytes of the location defined by the operand expression generate into register address references. The value of the expression must be an address and must not contain forward references.

EXAMPLE

LABEL	OPERATION	OPERAND	COMMENT
WRKSPACE	BLOCK	32	
SYMX	EQU	WRKSPACE+2	
SYMY	EQU	WRKSPACE+4	
	WPNT	WRKSPACE	;DEFINE CURRENT WORKSPACE
SUB	LWPI	WRKSPACE	;LOAD WORKSPACE POINTER
	INC	WRKSPACE	;INCREMENT R0
	MOV	SYMY+4,SYMX	;MOVE CONTENTS OF R4 TO ;R1

LOCATION COUNTER CONTROL DIRECTIVE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ORG	{[/] expression}	[;charstring]

PURPOSE

The ORG directive sets the contents of the assembler location counter to either the address specified by the operand expression, the next address divisible by the operand expression, or the next odd address.

EXPLANATION

Omission of the optional / (slash) operator sets the location counter to the address specified by the operand expression. For example, when the following ORG directive is entered, the next instruction in the program begins at location 11ØØH in the current section.

```
ORG 11ØØH
```

If an ORG directive is omitted at the beginning of a program, the assembler location counter is set to Ø. Usage of the / operator in the operand field causes the location counter to be set to the next location divisible by the operand expression. For example, when the current location counter contains 11ØØH and the following ORG directive is entered, the next instruction begins at location 111H. (The next location divisible by 15H is 111H).

```
ORG  /15H
```

If the current location counter is divisible by the operand condition when the / operator is present, the location counter is unaffected. If the operand expression is "/Ø", the location counter is set to the next odd value. For example, when the current location counter contains 11ØØH, and the following ORG directive is entered, the next instruction begins at location 11Ø1H.

```
ORG    /Ø
```

If the current location counter is already set to an odd value when the "/Ø" operand is entered, the location counter is unaffected.

The optional / operator may be used only with scalar-valued operand expressions.

Use care when entering the / operator, since the expected results may not be retained upon linking. For example, if ORG /Ø is entered, and the linker puts the section containing this directive on an odd address, the ORG result is on an even address. This problem can be corrected by using the LOCATE command in the Linker. (Refer to Section 9, THE LINKER.)

Any symbol contained in the operand expression must have been defined in the label field of a previous statement in the program. If the operand expression contains a symbol previously defined in the label field of an EQU directive, the operand field of that EQU directive must not contain forward-referenced symbols.

A label symbol is generally not entered with this statement; however, if used, the symbol represents the resulting value of the location counter.

The ORG directive should be used to locate instructions on even-numbered addresses. The assembler will correctly assemble instructions with odd-numbered addresses, but will cause an error code to appear.

EXAMPLE

The following ORG statement causes the object code generated by the next instruction to begin at location 1100H.

```
LABEL     OPERATION     OPERAND     COMMENT
.
.
.
          ORG           1100H       ;STARTS OBJECT CODE OF
                                    ;NEXT INSTRUCTION AT 1100H
L1        MOV           R1,R2       ;LOADS THE CONTENTS OF R1
                                    ;INTO R2
.
.
.
```

Upon assembly, the listing lines for the preceding instructions appear as follows. The MOV instruction object code begins at location 1100H. Notice the relocation indicator (>) on line 00008.

```
.
.
.
00008     1100    >         ORG  1100H      ;STARTS OBJECT CODE OF
00009                                       ;NEXT INSTRUCTION AT 1100H
00010     1100 C081   L1    MOV  R1,R2      ;LOADS THE CONTENTS OF R1
                                            ;INTO R2
.
.
.
```

DATA STORAGE CONTROL DIRECTIVES

The assembler data storage control directives appear in the order shown in the following summary.

Mnemonic	Purpose
BYTE	Allocates one byte of memory to each expression specified in the operand field.
WORD	Allocates two bytes of memory to each expression specified in the operand field.
ASCII	Stores ASCII text in memory.
BLOCK	Reserves a specified number of bytes in memory.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BYTE	{expression} [,expression] . . .	[;charstring]

PURPOSE

This directive allocates one byte of memory to each expression specified in the operand field.

EXPLANATION

Each data byte is represented by an expression. The data is stored in the object module in the order in which it appears in the operand field. If more than one expression is specified in the operand field, the expressions are stored in consecutive bytes. The optional label field symbol represents the address of the first byte of data specified by the directive.

If the expression represents a value exceeding the eight-bit capacity, the eight least significant bits are used and a truncation error code is displayed. For example, a statement containing the following BYTE directive generates 32H upon assembly and issues a truncation error response.

```
LABEL     OPERATION     OPERAND     COMMENT
          BYTE          "K2"        ;GENERATES 32H,
                                    ;TRUNCATION ERROR
```

EXAMPLE

In the following BYTE directive, one byte of memory is allocated to the expression values 24 hexadecimal and 22 decimal. The label symbol, FSTBYT, represents the address of the first byte specified, 24H.

```
LABEL     OPERATION     OPERAND      COMMENT
FSTBYT    BYTE          24H,22       ;ALLOCATES ONE BYTE OF
                                     ;MEMORY TO THE
                                     ;EXPRESSION VALUES 24H
                                     ;AND 22 DECIMAL
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	WORD	{expression} [,expression] . . .	[;charstring]

PURPOSE

The WORD directive allocates two bytes of memory to each expression specified in the operand field.

EXPLANATION

This directive is identical to the BYTE directive except that two bytes of memory are allocated in the object module for every expression specified in the operand field. These two-byte values are stored in memory with the high byte first, followed by the low byte. If an expression represents a single byte value, the high byte is stored as zero. If more than one expression is specified in the operand field, the expressions are stored in consecutive words. The optional label field symbol represents the address of the first byte of data stored in memory.

EXAMPLE

In the following WORD directive, two bytes of memory are allocated to the expression values 356 and 427 decimal. The label symbol LABSYM represents the address of the first byte of the value 356 decimal.

```
LABEL     OPERATION     OPERAND       COMMENT
LABSYM    WORD          356,427       ;ALLOCATES TWO BYTES OF
                                      ;MEMORY EACH TO THE
                                      ;EXPRESSION VALUES 356 AND
                                      ;427 DECIMAL
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ASCII	{string expression} [,string expression] . . .	[;charstring]

PURPOSE

The ASCII directive allows the user to store text in memory easily.

EXPLANATION

ASCII characters may be specified in the operand field in the form of a string expression. If more than one operand is specified on a line, each operand is separated by a comma. The optional label symbol represents the memory address allocated to the first operand field character.

EXAMPLES

Assume the following lines of source code reside on disc:

```
LABEL   OPERATION   OPERAND                      COMMENT
.
.
.
        ASCII       "HELLO", "GOODBYE"           ;PUTS HELLO AND
                                                 ;GOODBYE IN OBJECT
                                                 ;MODULE AS ASCII
                                                 ;CODE
        ASCII       "BYE"                        ;PUTS BYE IN OBJECT
                                                 ;MODULE AS ASCII
                                                 ;CODE
        ASCII       ""                           ;PUTS NULL STRING
                                                 ;IN OBJECT MODULE
                                                 ;AS ASCII CODE
        STRING      STR1 (20)                    ;DEFINES STR1 AS
                                                 ;STRING VARIABLE
                                                 ;WITH A MAXIMUM
                                                 ;CHARACTER LIMIT
                                                 ;OF 20
STR1    SET         "ABCDEF"                     ;ASSIGNS ASCII
                                                 ;VALUE OF ABCDEF
                                                 ;TO STR1
        ASCII       STR1                         ;PUTS ABCDEF IN
                                                 ;OBJECT MODULE AS
                                                 ;ASCII CODE
        ASCII       STR1:" ":STRING(NCHR(STR1))  ;PUTS ABCDEF, A
                                                 ;BLANK, AND THE
                                                 ;NUMBER OF
                                                 ;CHARACTERS IN
                                                 ;ABCDEF (6) IN
                                                 ;OBJECT MODULE AS
                                                 ;CONCATENATED
                                                 ;ASCII CODE
.
.
.
```

The hexadecimal object code generated by the string expressions in the preceding source code is shown as follows.

SOURCE	OBJECT
"HELLO", "GOODBYE"	48454C4C4F474F4F44425945
"BYE"	425945
""	(nothing)
"ABCDEF" (string value of STR1)	414243444546
"ABCDEF 000006"	41424344454620303030303036

For hexadecimal and ASCII conversion tables, refer to Appendix E.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BLOCK	{expression}	[;charstring]

PURPOSE

The BLOCK directive reserves a specified number of bytes in memory.

EXPLANATION

The BLOCK operand expression indicates the number of bytes to reserve in memory. The operand expression must be a positive value. The operand expression must be either a numeric or string constant, or a symbol. If the operand expression contains a symbol, the symbol must be previously defined in the program. Additionally, if the symbol is defined by the EQU directive, that EQU directive's operand field must conform to these same rules. The expression specified in the BLOCK operand must be a scalar value.

EXAMPLE

The following BLOCK directive reserves a 32-byte memory storage block:

LABEL	OPERATION	OPERAND	COMMENT
	BLOCK	32	;RESERVES 32 BYTES OF ;MEMORY

MACRO DEFINITION DIRECTIVES

The macro definition directives are presented in the order shown in the following summary. A complete description of macro capability is presented in Section 5.

Mnemonic	Purpose
MACRO	Defines the name of a source code block used repeatedly within a program.
ENDM	Terminates the macro definition block.
REPEAT	Enables the macro lines following the REPEAT statement up to the ENDR statement to be assembled repeatedly.
ENDR	Signals the corresponding REPEAT block termination.
INCLUDE	Inserts text from a specified file into the program.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	MACRO	{symbol}	[;charstring]

PURPOSE

The MACRO directive defines the name of a source code block used repeatedly within a program.

EXPLANATION

A macro is a shorthand method for inserting a block of source code into a program one or more times. The MACRO directive names the source code block to be inserted into the main program. The symbolic macro name appears in the operand field of the MACRO directive, and is later used as a reference when the source code block is called for insertion during assembly. The block of source code to be inserted is called the macro definition block, and immediately follows the MACRO directive. The macro definition block terminates with an ENDM directive. When the macro name appears within the operation field of the main program during assembly, the macro definition block is inserted and assembled within the main program. This process is called macro expansion.

The symbolic macro name and the macro definition block are generally defined at the beginning of a user program. The macro name and definition block must be defined prior to the initial macro definition block usage.

For a further description of macro capability and usage, refer to Section 5.

EXAMPLE

The MACRO directive below defines the block of macro code following the directive.

```
LABEL   OPERATION   OPERAND     COMMENT
        MACRO       MACRNAME    ;DEFINES MACRNAME AS MACRO
                                ;NAME
        BYTE        3,5,1       ;ALLOCATES ONE BYTE OF
                                ;MEMORY EACH TO THE CONSTANT
                                ;VALUES 3, 5, AND 1
        WORD        2           ;ALLOCATES TWO BYTES OF
                                ;MEMORY TO THE CONSTANT
                                ;VALUE 2
        ENDM                    ;END OF MACRO DEFINITION,
                                ;MACRNAME
        .
        .
        .
```

Later statements in this program may call the macro definition block whenever the specified BYTE and WORD statement sequence is desired.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ENDM		[;charstring]

PURPOSE

The ENDM directive signals the end of a macro definition block.

EXPLANATION

When an ENDM directive is encountered in a macro definition block, the macro is terminated and assembly continues with the next statement in the program following the macro call.

EXAMPLE

The following ENDM directive terminates the macro definition block named NUMNAK.

```
LABEL   OPERATION   OPERAND     COMMENT
        MACRO       NUMNAK      ;DEFINES NUMNAK AS MACRO
                                ;NAME
        BYTE        3,27,22     ;ALLOCATES ONE BYTE OF
                                ;MEMORY TO THE CONSTANT
                                ;VALUES 3, 27, AND 22
        WORD        255         ;ALLOCATES TWO BYTES OF
                                ;MEMORY TO THE CONSTANT
                                ;VALUE 255
        ENDM                    ;END OF MACRO DEFINITION
```

SYNTAX

LABEL	OPERATION	OPERAND		COMMENT
[symbol]	REPEAT	{expression1}	[,expression2]	[;charstring]
[symbol]	ENDR			[;charstring]

PURPOSE

The REPEAT directive enables the macro lines following the REPEAT directive, up to the ENDR directive, to be assembled repeatedly. The ENDR directive signals the end of each repeat cycle.

EXPLANATION

When a REPEAT directive is encountered upon macro expansion, the first expression specified in the operand field is evaluated. The lines up to the ENDR directive are ignored when the REPEAT operand, "expression1" is equal to zero (false). If the expression is true (non-zero), the lines up to the ENDR directive are assembled repeatedly until the expression does equal zero, or the maximum number of repeat cycles is exceeded. The second operand, "expression2" may be optionally entered to specify the maximum number of repeat cycles. If the maximum number of repeat cycles is not specified, the value of "expression2" defaults to 255. Attempts to repeat beyond the value of "expression2" cause an error code to be displayed. Both operand expressions must be scalar values.

REPEAT – ENDR blocks may be nested. The nesting depth is limited only by the amount of memory available to the assembler. Each REPEAT condition must be properly nested, thus having a matching ENDR occurring within the scope of that particular REPEAT condition. REPEAT – ENDR blocks may not cross the boundary of a macro expansion or of an IF – ENDIF block. A REPEAT – ENDR block is valid only within a macro definition block.

EXAMPLE

The example that follows demonstrates REPEAT – ENDR block usage within a macro named CONDRID.

```
LABEL   OPERATION   OPERAND      COMMENT
        MACRO       CONDRID      ;DEFINES CONDRID AS MACRO
                                 ;NAME
AGAIN   SET         1            ;INITIALIZES AGAIN TO EQUAL
                                 ;1 AT ASSEMBLY TIME
        REPEAT      AGAIN <= 27  ;REPEAT WHILE AGAIN IS LESS
                                 ;THAN OR EQUAL TO 27
        BYTE        AGAIN        ;GENERATES ONE BYTE OF
                                 ;MEMORY TO AGAIN
AGAIN   SET         AGAIN + 1    ;INCREMENT AGAIN AT
                                 ;ASSEMBLY TIME
        ENDR                     ;END OF REPEAT CONDITION
        BYTE        ØDH          ;GENERATES CARRIAGE
                                 ;RETURN
        ENDM                     ;END OF MACRO DEFINITION
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	INCLUDE	{string expression}	[;charstring]

PURPOSE

The INCLUDE directive is used to insert text from a specified file into a program.

EXPLANATION

When the INCLUDE directive is encountered, text from the file specified in the operand field is inserted into the program. If the INCLUDE directive is contained in a macro body, the text file is inserted at macro expansion time. Parameters within the included file cannot reference arguments used in the containing macro. Refer to Section 5 for a discussion of text substitution within macros. The text file specified by the INCLUDE directive may not terminate a MACRO, REPEAT or IF block. Additionally, the text may not contain another INCLUDE directive.

An INCLUDE directive may also be used within normal source code, outside of macro definition blocks. When this occurs, the inserted text may contain macro definitions.

EXAMPLE

The following example demonstrates INCLUDE directive usage.

```
LABEL    OPERATION    OPERAND      COMMENT
              .
              .
              .
         INCLUDE      "OTHFILE"    ;INSERTS OTHFILE INTO THE
              .                     ;CURRENT PROGRAM AT THE
              .                     ;ADDRESS OF THE CURRENT
              .                     ;LOCATION COUNTER.
```

CONDITIONAL ASSEMBLY DIRECTIVES

The conditional assembly directives are presented in the order shown in the following summary.

Mnemonic	Purpose
IF	Causes the assembly of the source code lines following the IF directive, up to the ENDIF directive, when the specified operand expression is true (non-zero).
ELSE	Causes an alternate source block to be assembled when the containing IF expression is false.
ENDIF	Signals the corresponding IF block termination.
EXITM	Terminates the current macro expansion before encountering an ENDM directive.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	IF	{expression}	[;charstring]
[symbol]	ELSE		[;charstring]
[symbol]	ENDIF		[;charstring]

PURPOSE

The IF directive causes assembly of the source code lines following the IF directive, up to the ENDIF (or ELSE, if present) directive, when the specified operand expression is true. The ELSE directive causes an alternate source block to be assembled when the containing IF expression is false. ENDIF signals the corresponding IF block termination.

EXPLANATION

When an IF directive is encountered, the expression specified in operand field is evaluated. If the result of the expression is zero (false), source lines between the IF and ENDIF directives are ignored (not assembled). The ENDIF directive then terminates the condition. If the result of the expression is non-zero (true), the source lines are assembled once normally.

An optional ELSE directive block may be nested within the IF source block. If an ELSE block is present, a false IF expression causes assembly of the source lines from the ELSE directive up to the ENDIF directive. The ELSE block is ignored when the expression in the IF directive operand field is true. Only one ELSE directive is allowed within each IF – ENDIF block.

IF – (ELSE) – ENDIF blocks may be nested as deeply as desired, limited only by the amount of memory available to the assembler. Each IF directive must be properly nested, thus having a matching ENDIF occurring within the scope of that particular IF condition. IF – (ELSE) – ENDIF blocks may not cross the boundaries of REPEAT – ENDR blocks, macro expansions, and other IF – (ELSE) – ENDIF blocks.

EXAMPLES

The following example demonstrates IF – (ELSE) – ENDIF block usage:

```
LABEL   OPERATION   OPERAND      COMMENT
        IF          "'1'" = ""   ;CHECKS TO SEE IF THE FIRST
                                 ;MACRO ARGUMENT IS
                                 ;UNDEFINED
        WORD        ØF7H         ;IF SO, GENERATES A WORD
                                 ;CONTAINING ØF7H
        ELSE                     ;OTHERWISE
        WORD        '1'          ;GENERATES A WORD
                                 ;CONTAINING THE FIRST
                                 ;ARGUMENT
        ENDIF                    ;END OF IF CONDITION
```

The following example demonstrates nested IF – (ELSE) – ENDIF block usage:

```
LABEL   OPERATION   OPERAND      COMMENT
        IF          "'1'"<>""    ;CHECKS TO SEE IF THE FIRST
                                 ;MACRO ARGUMENT
                                 ;EXISTS
        IF          '1'< ØFØH    ;IF SO, CHECKS TO SEE IF THE
                                 ;FIRST MACRO ARGUMENT IS
                                 ;LESS THAN ØFØH
        WORD        ØF7H – '1'   ;IF SO, GENERATES ONE WORD
                                 ;CONTAINING THE DIFFERENCE
                                 ;BETWEEN ØF7H AND THE
                                 ;FIRST ARGUMENT
        ELSE                     ;OTHERWISE, IF FIRST
                                 ;ARGUMENT IS GREATER
                                 ;THAN ØFØH. . .
```

(Program continued on next page)

LABEL	OPERATION	OPERAND	COMMENT
	WORD	'1'	;GENERATES ONE WORD
			;CONTAINING FIRST MACRO
			;ARGUMENT
	ENDIF		;END OF INNER IF CONDITION
	ELSE		;OTHERWISE, IF THE
			;ARGUMENT DOES NOT EXIST. . .
	WORD		;GENERATE A WORD
			;CONTAINING ØF7H
	ENDIF		;END OF OUTER IF CONDITION

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	EXITM		[;charstring]

PURPOSE

The EXITM directive terminates the current macro expansion before encountering an ENDM directive.

EXPLANATION

EXITM is generally used within IF – (ELSE) – ENDIF and REPEAT – ENDR blocks to conditionally terminate macro expansions. EXITM is valid only within a macro definition block.

EXAMPLE

The following example demonstrates conditional macro termination with the EXITM directive.

```
LABEL   OPERATION   OPERAND     COMMENT
        MACRO       CONDMAC     ;DEFINES CONDMAC AS MACRO
                                ;NAME
        BYTE        1,2,Ø       ;ALLOCATES ONE BYTE OF
                                ;MEMORY FOR EACH OF THE
                                ;THREE VALUES 1, 2, AND Ø
        IF          "'3' "=""   ;TESTS TO DETERMINE IF
                                ;3RD PARAMETER IN
                                ;MACRO CALL EXISTS
        BYTE        255         ;IF 3RD ARGUMENT DOES NOT
                                ;EXIST, ONE BYTE IS ALLOCATED
                                ;CONTAINING 255 DECIMAL
        EXITM                   ;TERMINATES MACRO
                                ;EXPANSION IF CONDITION IS
                                ;SATISFIED
        ENDIF                   ;END OF IF CONDITION
        BYTE        '3'         ;OTHERWISE, ONE BYTE IS
                                ;ASSIGNED CONTAINING THIRD
                                ;ARGUMENT
        ENDM                    ;END OF MACRO DEFINITION
```

SECTION DEFINITION DIRECTIVES

The section definition directives appear in this subsection in the order shown in the summary below. The ABSOLUTE option, which may be used with the SECTION and COMMON directives, is described following this summary. For a discussion of the methods by which the Linker relocates sections, refer to Section 9, THE LINKER.

Mnemonic	Purpose
SECTION	Declares a Linker section, assigns a section name, and defines the section parameters.
COMMON	Declares a Linker section, assigns a section name, and defines the section type to be common.
RESERVE	Sets aside a work space in memory. Upon linking, all reserve sections with the same name are concatenated into a single reserve section.
RESUME	Continues the definition of code for a given section.
GLOBAL	Declares one or more symbols to be global variables.
NAME	Declares the name of an object module.

RELOCATION OPTION

The ABSOLUTE option may be specified in the operand field of the SECTION and COMMON directives. This option causes the memory allocation to be the actual area specified by the ORG directive at assembly time. (No relocation of this section is performed.) When this option is not specified, the section is relocated on any even-numbered byte address (word relocatable).

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	SECTION	{symbol} [ABSOLUTE]	[;charstring]

PURPOSE

The SECTION directive is used to declare a program section, assign the section a name, and define its parameters.

EXPLANATION

All program text following the SECTION directive, up to the next SECTION, COMMON, or RESUME directive, is defined to be a program section. All text within a program section is assembled with the same location counter, and hence, has the same base. Each section has a separate location counter and must be relocated as a block. The initial value of the location counter for a given section is Ø. The symbol specified in the SECTION operand field is the section name, and is a global symbol. The section name must be unique to each assembly and, therefore, cannot appear in multiple SECTION directives. When separate object modules containing sections with the same name are linked, an error is generated.

The optional second operand (,ABSOLUTE) in the SECTION directive may be used to prevent text block relocation. (Refer to previous discussion on Absolute Option in this subsection.) If no option is specified, the program text block may be relocated on any even address (word) boundary.

When a label symbol is entered on the SECTION directive, the symbol represents address Ø, the initial value of the resulting section's location counter. Additionally, the declared section name in the operand field may be used as a normal global symbol, and referenced in the operand field of other statements throughout the assembly. The section name has the same value as the label on the SECTION directive.

EXAMPLE

The following source line demonstrates SECTION directive usage.

```
LABEL     OPERATION     OPERAND          COMMENT
          .
          .
          .
          SECTION       SEC1             ;GENERATES WORD-
          .                              ;RELOCATABLE SECTION,
          .                              ;SEC1
          .
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	COMMON	{symbol} [ABSOLUTE]	[;charstring]

PURPOSE

The COMMON directive declares a section, associates a name with the section, assigns the section parameters, and defines the section type to be common.

EXPLANATION

The COMMON directive performs the same functions as the SECTION directive, except that the same name may identify common sections in more than one source module. Common sections with the same name are relocated at the same address by the Linker. Each section with the same name should specify the same relocation type; otherwise, the desired relocation might not result at link time. The Linker allocates enough memory to contain the largest of the common sections with the same name.

This section type is modeled after the COMMON area of FORTRAN.

EXAMPLE

The following example demonstrates COMMON directive usage.

```
LABEL   OPERATION   OPERAND     COMMENT
        .
        .
        .
        COMMON      WRKAREA     ;DEFINES WRKAREA AS A
        .                       ;COMMON SECTION. IF
        .                       ;WRKAREA EXISTS IN
        .                       ;MULTIPLE OBJECT MODULES,
                                ;LINKER CHOOSES THE
                                ;LARGEST SECTION NAMED
                                ;WRKAREA FOR MEMORY
                                ;ALLOCATION.
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	RESERVE	{symbol, expression}	[;charstring]

PURPOSE

The RESERVE directive is used to set aside a workspace in memory. Upon linking, all reserved workspaces (sections) with the same name are combined into a single section.

EXPLANATION

The symbol in the operand field of the RESERVE directive is the assigned name of the section. The operand expression specifies the number of bytes to be reserved for the current object module. The expression must be a scalar value. The RESERVE directive does not change the current section.

More than one object module may contain reserve sections of the same name. The length of the reserve section allocated by the Linker is the sum of all reserve sections with the same name.

EXAMPLE

The following example demonstrates section space allocation with the RESERVE directive.

```
LABEL     OPERATION     OPERAND              COMMENT
                .
                .
                .
          RESERVE       BNCHCODE,100H        ;RESERVES A SECTION DEFINED
                                             ;AS BNCHCODE AND
                                             ;ALLOCATES 256 BYTES OF
                                             ;MEMORY TO BE ADDED TO THE
                                             ;SIZE OF BNCHCODE
                .
                .
                .
          WORD          BNCHCODE             ;PLACES ONE WORD IN THE
                                             ;CURRENT SECTION HAVING
                                             ;THE ADDRESS OF THE
                                             ;BEGINNING OF THE BNCHCODE
                                             ;SECTION
          WORD          ENDOF(BNCHCODE)      ;PLACES ONE WORD IN THE
                                             ;CURRENT SECTION HAVING
                                             ;THE ENDING ADDRESS OF
                                             ;BNCHCODE
                .
                .
                .
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	RESUME	[symbol]	[;charstring]

PURPOSE

The RESUME directive continues the definition of a given section.

EXPLANATION

The RESUME directive continues the definition of the section specified by the optional operand symbol. If no operand symbol is used, the definition of the default section is continued. Any source code that is not preceded by a SECTION or COMMON directive is included in the default section. The name given to the default section is a percent sign (%) followed by the object file name. When no object file is present, the name given to the default section is %.

If used, the label symbol is assigned the value of resumed section's location counter.

EXAMPLE

The example that follows demonstrates section definition resumption with the RESUME directive.

```
LABEL   OPERATION   OPERAND   COMMENT
            .
            .
        SECTION     A31       ;DEFINES SECTION A31
            .
            .
            .
        SECTION     B31       ;DEFINES SECTION B31
            .
            .
            .
        RESUME      A31       ;RESUMES SECTION A31
            .
            .
            .
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	GLOBAL	{symbol} [,symbol] . . .	[;charstring]

PURPOSE

The GLOBAL directive declares one or more symbols to be global variables. A global variable located in one source module may be referenced by another source module.

EXPLANATION

Symbols specified in the GLOBAL directive operand field are designated to be global variables. Global variables defined in the current assembly are called bound globals. If the global variables are not defined in the current assembly, they are called unbound globals and their references must be resolved by the Linker.

The value of a global symbol must be unique within an assembly. A maximum of 254 names may be defined to be global variables. This maximum includes all names used in SECTION, COMMON, RESERVE, and GLOBAL directives. When the default section is used, the names in the default section are also counted toward the maximum.

EXAMPLE

The following example demonstrates definition of global variables with the GLOBAL directive.

```
LABEL     OPERATION     OPERAND          COMMENT
                .
                .
                .
          GLOBAL        HIGUY,BYEGUY     ;DEFINES THE SYMBOLS HIGUY
                                         ;AND BYEGUY TO BE USED AS
                                         ;GLOBAL SYMBOLS
                .
                .
                .
HIGUY     EQU           $                ;HIGUY IS EQUIVALENT TO
                                         ;CURRENT LOCATION
                                         ;COUNTER
          BL            BYEGUY           ;JUMPS TO SUBROUTINE
                                         ;BYEGUY DEFINED IN
                                         ;ANOTHER ASSEMBLY
                .
                .
                .
```

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	NAME	{symbol}	[;charstring]

PURPOSE

The NAME directive declares the name of an object module.

EXPLANATION

The symbol in the operand field of the NAME directive is the name assigned to the object module. If more than one NAME directive appears within an assembly, only the first NAME directive is used; the rest are ignored.

Note that the object module name, as declared by the NAME directive, is distinct from the file name that the object module is stored under. Note also that the default section derives its name from the object file, not the NAME directive.

EXAMPLE

The following example demonstrates object module naming with the NAME directive.

```
LABEL     OPERATION     OPERAND          COMMENT
            .
            .
            .
          NAME          "XMPLSUB"        ;NAMES OBJECT MODULE
                                         ;XMPLSUB
            .
            .
            .
```

MODULE TERMINATION DIRECTIVE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	END	[expression]	[;charstring]

PURPOSE

The END directive terminates source modules.

EXPLANATION

The END directive terminates a source module contained in one or more disc files. A source module is also terminated when the end of the last input file is read. END directive usage is, therefore, optional.

The optional expression in the operand field represents the starting address for program execution, which is called a transfer address. If present, the specified operand value is placed in the object module and may be used by the TEKDOS LOAD command when loading the object module into program memory. At link time, if more than one module has a transfer address, the first one encountered is used.

Section 5

MACROS

INTRODUCTION

A macro is a shorthand approach for inserting source code into a program. A macro is often used when the same, or nearly the same, code is repeatedly used within a program. A block of macro code is called a macro definition block. The source code that results from this block may be altered each time the macro is called so that the object code generated depends on the information specified in the macro call. The code generated by a macro call is called a macro expansion, since it results from, and is usually larger than, the macro call.

This section describes all phases of macro definition, calling, and expansion. The structure of this section closely follows the process leading up to macro expansion. First, an examination of the general macro expansion process is illustrated to provide a basis of understanding. An examination of each phase of the process is then presented in greater detail.

BASIC MACRO EXPANSION PROCESS

The macro expansion process is illustrated in Fig. 5-1. A written explanation of the process follows the figure.

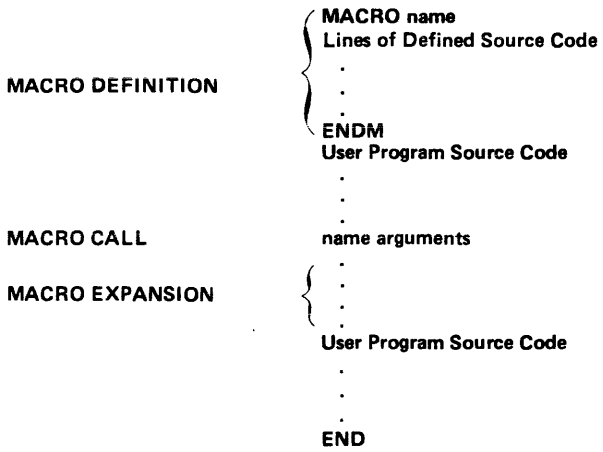


Fig. 5-1. The Macro Expansion Process. 2415-4

As mentioned, there are three phases of macro usage: definition, calling, and expansion. First the macro must be defined. The macro is given a name followed by a body. The macro is defined in a macro definition directive. The macro body is called a macro definition block. The macro definition block is made up of source lines that are stored in unassembled form, until the macro is used. To use the macro, the programmer codes a macro call within a program. The macro name appears in the macro call directive's operation field. When the macro call is encountered during assembly, the macro definition block is inserted and assembled within the main program. This process is called macro expansion.

The user may alter any parameters used within the macro definition block by inserting corresponding arguments within the operand field of a macro call. One line at a time, the assembler replaces the specified parameters with corresponding arguments in the macro call. The assembler inserts the line from the macro definition block into the user program. The line is then assembled. This procedure repeats for each line in the macro definition block.

MACRO DEFINITION DIRECTIVE

A macro is defined by first entering the macro definition directive in the following format. In this macro definition directive, "name" is the macro name that is later used as a reference for the macro call.

MACRO name

Macro Definition Directive Conventions

A macro is generally defined at the beginning of a program. A macro must always be defined prior to its initial use. A macro may not be defined within another macro definition block. A macro name is a symbol containing up to eight characters, the first character being alphabetic. The macro name must be unique from all symbols in a user program.

MACRO DEFINITION BLOCK

The lines following the macro definition directive, up to and including an ENDM directive, become a pre-defined block of code referred to as a macro definition block. A macro definition block may contain any instruction or assembler directive (except the END and MACRO directives). A macro definition block may contain calls to other macros or even calls to itself. When a macro call occurs within another macro definition block, any replacement that may occur on the macro call is performed before the inner macro is called. A macro definition block may not contain the definition of another macro.

Source Code Alteration

An additional macro capability allows code to be altered within a macro definition block. Upon expansion, parameters within single quotes, serving as place holders in the macro definition block, are replaced by the arguments defined in a macro call.

In summation:

Parameters	— are place holders within a macro-definition block.
Arguments	— are values, defined within a macro call directive, that replace parameters.

Any numeric parameter surrounded by single quotes ('N') is replaced by the Nth argument passed to the current macro expansion. In the following BYTE directive, for example, the first argument passed to the current macro expansion is substituted for the first parameter, labeled '1', upon macro expansion.

```
BYTE 3,5,'1'
```

N may be either a number or a numeric-valued SET symbol. A SET symbol is assigned a value by the SET directive. This capability is discussed in Section 4, ASSEMBLER DIRECTIVES, describing the SET directive. If N is greater than the number of arguments provided, the null string is substituted. Text substitution may occur anywhere on a line.

Additional Special Macro Definition Characters

The following special characters are only available for use within macro definition blocks.

The @ Character

The "at" character, when surrounded by single quotes ('@'), provides unique labels for each macro expansion. The @ character is replaced by a four-character hexadecimal value that is unique within each macro call. In the example that follows, each time the macro is called, a unique four-character hexadecimal value replaces the @ character. The following statement creates a unique seven-character label.

```
LABEL     OPERATION     OPERAND
LAB '@'   EQU           $
```

The '@' in the preceding label is replaced by a number unique to the current macro call. This replacement prevents LAB from being defined more than once by subsequent macro calls.

The # Character

The "pound" character, when surrounded by single quotes ('#'), is replaced by a five-digit decimal number. The number represents the total number of arguments that are passed to the current macro expansion. In the example that follows, expansion of all lines of code within a REPEAT block continues until the total number of arguments passed is exceeded. Suppose three arguments are passed during expansion of the macro containing this code:

```
LABEL   OPERATION   OPERAND     COMMENT
                    .
                    .
                    .
J       SET         1           ;INITIALIZES J TO EQUAL 1
                                ;AT ASSEMBLY TIME
        REPEAT      J <= '#'    ;REPEAT WHILE J IS LESS THAN
                                ;OR EQUAL TO 3
J       SET         J + 1       ;INCREMENT J
                    .           .
                    .           .
        ENDR                    ;END OF REPEAT CONDITION
                    .
                    .
                    .
```

The % Character

The "percent" character, when surrounded by single quotes ('%'), is replaced by the name of the current section or common. The name is returned as a string. If the current section is the default section, the null string is returned.

In the example that follows, the percent sign character is used to represent the name of the current section.

```
LABEL     OPERATION     OPERAND          COMMENT
          STRING        SECNAM(8)        ;DEFINES STRING, SECNAM,
                                         ;WITH EIGHT-CHARACTER
                                         ;MAXIMUM
SECNAM    SET           " '%' "          ;SECNAM IS SET TO NAME OF
                                         ;CURRENT SECTION
          SECTION       BBB              ;DEFINES NEW SECTION BBB
          .
          .
          .
          RESUME        'SECNAM'         ;RESUMES PREVIOUS SECTION
          .
          .
          .
```

The ↑ or ^ Character In Macro Definition

The up-arrow (↑) or caret (^) character may be entered just prior to any character having special meaning, thus allowing the special character to be interpreted as a regular part of the text. The ↑ or ^ is available in all phases of the TEKTRONIX Assembler and is described in the manner in which it affects macro definition. In the example that follows, the caret (^) character removes the special meaning of the single quote character.

```
LABEL     OPERATION     OPERAND
          ASCII         "THAT^'S ALL FOLKS."
```

Upon macro expansion, the following code is generated in memory:

```
          THAT'S ALL FOLKS.
```

MACRO TERMINATION

A macro definition block is terminated by an ENDM statement.

MACRO CALLING

A macro is invoked when a macro call is encountered within a program. A macro call contains the macro name to be called in the statement's operation field as follows:

LABEL	OPERATION	OPERAND
	name	

INCLUDE Directive Text Insertion

Another method for calling text into a program involves INCLUDE directive usage. The INCLUDE directive (see Section 4, describing ASSEMBLER DIRECTIVES) may be used to insert text into a program from a specified file. The INCLUDE directive may be part of a MACRO, IF – ENDIF, or REPEAT – ENDR block, as long as it does not terminate any of those blocks. The name of the file to be inserted is entered in the operand field of the INCLUDE directive as follows:

LABEL	OPERATION	OPERAND
	INCLUDE	"filename"

Text Substitution

Optional arguments separated by commas within the operand field of the macro call define the values to replace the parameters within the block as the macro is expanded. For example, the following macro call invokes the macro named EVALC and defines the arguments 25 and ARG2 for substitution within the block of code as the macro is expanded.

```
LABEL    OPERATION    OPERAND      COMMENT
         EVALC        25,ARG2      ;INVOKES MACRO EVALC AND
                                   ;DEFINES FIRST TWO
                                   ;ARGUMENTS FOR
                                   ;SUBSTITUTION WITHIN MACRO
                                   ;DEFINITION BLOCK AS 25
                                   ;AND ARG2
```

The preceding example contains the following arguments:

Argument 1 = 25

Argument 2 = ARG2

A label appearing in a macro call is assigned the value of the location counter prior to macro expansion.

Special Macro Calling Characters

The following special function is available for use within macro calls.

The [ ] Construct

Square brackets [ ] may be used to group code for inclusion as an argument within a macro call. All characters enclosed within square brackets are considered to represent a single argument. Square brackets may not be nested. Unlike the argument resulting when a character string is enclosed within double quotes, the square brackets are not passed to the source text during macro expansion. For example, the following macro call parameters produce the corresponding arguments.

```
LABEL    OPERATION    OPERAND                  COMMENT
         PNPDG        ABC,1,"ABC,1",[ABC,1]    ;INVOKES MACRO
                                                ;PNPDG AND
                                                ;SUBSTITUTES THE
                                                ;ARGUMENTS ABC,
                                                ;1, "ABC,1", ABC,1
```

The preceding example contains the following arguments.

Argument 1 = ABC

Argument 2 = 1

Argument 3 = "ABC,1"

Argument 4 = ABC,1

The ↑ or ^ Character In Macro Calls

The up-arrow ↑ or caret ^ character may be entered just prior to any character having special meaning, thus allowing that character to be interpreted as a regular part of the text. The ↑ or ^ symbol is available in all phases of the TEKTRONIX Assembler and is described in the manner in which it affects macro calls. The example that follows allows the comma and square bracket characters, respectively, to be interpreted as part of the arguments SML,J and [BC] when the macro TIME is invoked.

```
LABEL   OPERATION   OPERAND              COMMENT
        TIME        1,2,SML^,J,^[BC^]    ;INVOKES MACRO TIME AND
                                         ;SUBSTITUTES THE
                                         ;ARGUMENTS
                                         ;1,2,SML,J, AND [BC]
```

The preceding example contains the following arguments.

Argument 1 = 1

Argument 2 = 2

Argument 3 = SML,J

Argument 4 = [BC]

Additional Macro Argument Conventions

Any leading or trailing blanks are removed from the argument upon macro expansion. Blanks inserted within an argument are retained. If there are only blanks between two commas, the resulting argument is empty. To force a parameter to be replaced by blanks, it may be enclosed within square brackets. Examples of these conventions follow.

```
LABEL   OPERATION   OPERAND
        PQRD        A,B, C ,,[ D,E ],'' '', [  ], [^[ ]
```

The preceding example expands to the following arguments. Asterisks are used only in this example to indicate the beginning and end of the argument and are not expanded as part of the macro text.

```
Argument 1 = *A*
Argument 2 = *B*
Argument 3 = *C*
Argument 4 = **
Argument 5 = *  D,E  *
Argument 6 = *'' ''*
Argument 7 = *   *
Argument 8 = *[*
```

Any number or length of arguments may be entered within the operand field of a macro call, as long as the line does not exceed 128 characters (not including a carriage return). In addition, after arguments are substituted for parameters, the lines resulting from the macro expansion must not exceed 128 characters. Otherwise, an error code is displayed.

EXAMPLES

The following text includes two examples of macro definition, calling, and the resulting expansions. The first example illustrates a simple macro expansion. The second example is more complex and illustrates two contiguous macro expansions, where one is referenced by the other.

Example 1

In this example, a macro is defined as EVALC. Two parameters, 1 and 2, are defined and surrounded by single quotes within the macro definition block.

```
LABEL   OPERATION   OPERAND     COMMENT
        MACRO       EVALC       ;DEFINES EVALC AS MACRO
                                ;NAME
        BYTE        5,'1'       ;ALLOCATES ONE BYTE OF
                                ;MEMORY FOR THE CONSTANT
                                ;VALUE 5 AND ONE BYTE FOR
                                ;THE FIRST PARAMETER
                                ;WITHIN EVALC
        WORD        '2'         ;ALLOCATES TWO BYTES OF
                                ;MEMORY FOR THE SECOND
                                ;PARAMETER WITHIN EVALC
        ENDM                    ;END OF MACRO DEFINITION
```

Assume the following call appears within a user program.

```
LABEL   OPERATION   OPERAND     COMMENT
        EVALC       25,357      ;INVOKES MACRO EVALC AND
                                ;SUBSTITUTES THE
                                ;ARGUMENTS 25 AND 357 FOR
                                ;THE FIRST TWO
                                ;PARAMETERS WITHIN EVALC
```

This macro call generates the following macro expansion and substitutes the arguments 25 and 357 for the first two parameters ('1' and '2') within the macro definition block. The argument 357 requires two bytes of memory as defined by the WORD statement within the macro definition block.

```
LABEL   OPERATION   OPERAND
        BYTE        5,25
        WORD        357
```

Example 2

In the following example, two macro definition blocks are sequentially defined Q1 and Q2. One parameter is defined within each macro definition block. A macro call, Q1 7, is defined within Q2. This statement calls the macro, Q1.

```
LABEL   OPERATION   OPERAND     COMMENT
        MACRO       Q1          ;DEFINES Q1 AS MACRO NAME
PARM1   SET         1           ;ALLOWS SYMBOLIC REFERENCE
                                ;TO THE FIRST PARAMETER
        BYTE        3,5,'PARM1' ;ALLOCATES ONE BYTE OF
                                ;MEMORY EACH FOR THE
                                ;CONSTANT VALUES 3 AND 5,
                                ;AND FOR THE FIRST
                                ;PARAMETER PASSED TO Q1,
                                ;'PARM1'
        ENDM                    ;END OF MACRO DEFINITION Q1
        MACRO       Q2          ;DEFINES Q2 AS MACRO NAME
        BYTE        3,5,'1'     ;ALLOCATES ONE BYTE OF
                                ;MEMORY EACH FOR THE
                                ;CONSTANT VALUES 3 AND 5,
                                ;AND FOR THE FIRST
                                ;PARAMETER PASSED TO
                                ;Q2, '1'
        Q1          7           ;CALLS MACRO Q1 AND
                                ;ASSIGNS THE VALUE 7 TO THE
                                ;FIRST PARAMETER PASSED
                                ;TO Q1, 'PARM1'
        BYTE        8,9,10      ;ALLOCATES ONE BYTE OF
                                ;MEMORY EACH TO THE
                                ;CONSTANT VALUES 8, 9, AND
                                ;10
        ENDM                    ;END OF MACRO DEFINITION
                                ;Q2
```

Assume the following macro call appears within a user program to invoke the macro defined as Q2.

```
LABEL   OPERATION   OPERAND   COMMENT
        Q2          3         ;CALLS THE MACRO Q2 AND
                              ;SUBSTITUTES THE ARGUMENT
                              ;3 FOR THE FIRST PARAMETER
                              ;'1'
```

This macro call generates the following macro expansion.

```
LABEL   OPERATION   OPERAND
        BYTE        3,5,3
        BYTE        3,5,7
        BYTE        8,9,1Ø
```

In this example, the macro call Q2 3, causes the first statement within the macro Q2, BYTE 3,5,'1', to be expanded to BYTE 3,5,3. Expansion proceeds to the next statement that calls the macro Q1 and appears as Q1 7. This statement causes expansion to continue with the statement, PARM1 SET 1, thus allowing PARM1 to be used as a symbolic reference to the first parameter. This causes the next statement within Q1 to be expanded as BYTE 3,5,7, replacing BYTE 3,5,'PARM1'. Expansion within macro Q1 then terminates with the ENDM directive. This termination causes expansion to continue with the next statement in the referencing macro, Q2. The statement, BYTE 8,9,1Ø is the next statement that is expanded. Control then returns to the main program upon expansion of the ENDM directive, which terminates the macro expansion, Q2.

CONDITIONAL ASSEMBLY

Macros may be defined such that their expansion is conditional; that is, based upon the values of the parameters they use. IF – ELSE – ENDIF blocks allow conditional assembly and are valid in all phases of the TEKTRONIX Assembler. REPEAT – ENDR blocks also allow conditional assembly and are only valid within a macro definition. The two methods for performing conditional assembly are summarized as follows. For further information pertaining to IF – ELSE – ENDIF and REPEAT – ENDR usage, refer to Section 4, ASSEMBLER DIRECTIVES.

	OPERATION	OPERAND	
1)	IF	expr	Turns off the assembly process if the expression is equal to zero (false). Succeeding statements are passed over and are not acted upon until the ENDIF, or optional ELSE, statement is encountered.
	ELSE		Regenerates assembly process when IF expression equals zero. Usage is optional.
	ENDIF		Terminates the program text controlled by the corresponding IF statement.
2)	REPEAT	expr1,expr2	If expr1 is equal to zero (false), statements up to the ENDR statement are ignored. Otherwise, the statements are assembled and the assembler repeats the process again until the expression is equal to zero. A REPEAT block stops iterating when the specified expression maximum, expr2, is reached. If expr2 is not specified, the REPEAT block stops after 255 iterations.
	ENDR		Terminates the program text controlled by the corresponding REPEAT statement.

Nesting

IF – ELSE – ENDIF blocks and REPEAT – ENDR blocks may be nested. The nesting depth is limited only by the amount of memory available to the assembler. Each IF condition must be properly nested, having a matching ENDIF statement that occurs within the scope of that particular IF condition. Only one ELSE directive is permitted within each IF – ENDIF block. In addition, each REPEAT condition must be properly nested, having a matching ENDR statement occurring within the scope of that particular REPEAT condition. IF – ENDIF and REPEAT – ENDR blocks may not cross the boundary of a macro expansion or the boundaries of each other.

Conditional Macro Termination

The EXITM directive terminates the current macro expansion before the assembler encounters an ENDM directive. The EXITM directive is generally used within IF – ELSE – ENDIF and REPEAT – ENDR blocks to conditionally terminate macro expansions. EXITM is valid only within macro definition blocks.

EXAMPLES

IF—ENDIF Blocks

The following example demonstrates the definition, calling, and expansion of a macro using an IF – ENDIF block. The example also demonstrates the use of an EXITM directive

to conditionally terminate the macro expansion. In this example, a macro is defined as CONDIF and uses four parameters.

```
LABEL   OPERATION   OPERAND              COMMENT
        MACRO       CONDIF               ;DEFINES CONDIF AS MACRO
                                         ;NAME
        BYTE        '1','2',Ø,Ø,Ø        ;ALLOCATES ONE BYTE OF
                                         ;MEMORY FOR EACH OF FIVE
                                         ;VALUES. THE FIRST AND
                                         ;SECOND VALUES ARE THE
                                         ;FIRST AND SECOND
                                         ;PARAMETERS FOR
                                         ;SUBSTITUTION BY THE MACRO
                                         ;CALL ARGUMENTS. THE 3RD,
                                         ;4TH, AND 5TH VALUES ARE
                                         ;THE CONSTANT, Ø
        IF          '' '3' ''='''        ;TESTS 3RD PARAMETER TO
                                         ;DETERMINE IF IT EXISTS
        BYTE        255                  ;IF 3RD PARAMETER DOES NOT
                                         ;EXIST, ONE BYTE IS
                                         ;GENERATED CONTAINING
                                         ;255 DECIMAL
        EXITM                            ;TERMINATES MACRO
                                         ;EXPANSION, IF CONDITION
                                         ;IS SATISFIED
        ENDIF                            ;END OF IF CONDITION
        BYTE        '3'                  ;OTHERWISE, ONE BYTE IS
                                         ;ASSIGNED CONTAINING 3RD
                                         ;PARAMETER
        BYTE        HI('4'),LO('4')      ;SWAPS BYTES OF 4TH
                                         ;PARAMETER
        ENDM                             ;END OF MACRO DEFINITION
```

Assume the following macro call appears within a main program.

```
LABEL   OPERATION   OPERAND              COMMENT
        CONDIF      22,29,27,25          ;INVOKES MACRO CONDIF AND
                                         ;USES THE ARGUMENTS 22, 29,
                                         ;27, AND 25 FOR SUBSTITUTION
                                         ;OF THE FIRST FOUR
                                         ;PARAMETERS
```

This macro call substitutes the arguments 22, 29, 27, and 25 for the parameters labeled '1', '2', '3', and '4'. Notice that the substitution indicator (+) is displayed prior to each listed source line where substitution occurs.

```
0000 161D0000+      BYTE    22,29,0,0,0     ;ALLOCATES ONE BYTE OF
                                            ;MEMORY
0004 00
0005 1B       +     BYTE    27              ;OTHERWISE, ONE BYTE IS
                                            ;ASSIGNED
0006 0019     +     BYTE    HI(25),LO(25)   ;SWAPS BYTES OF 4TH
                                            ;PARAMETER
```

If the third substituted argument in this expansion had been empty rather than 27, the EXITM statement would have terminated further macro expansion.

REPEAT – ENDR Blocks

In the following example of a REPEAT – ENDR block, a macro is defined as CONDR and defines the SET symbol, AGAIN.

```
LABEL     OPERATION     OPERAND       COMMENT
          MACRO         CONDR         ;DEFINES CONDR AS MACRO
                                      ;NAME
AGAIN     SET           1             ;INITIALIZES AGAIN TO EQUAL
                                      ;1 AT ASSEMBLY TIME
          REPEAT        AGAIN <= '#'  ;REPEAT WHILE AGAIN IS LESS
                                      ;THAN OR EQUAL TO TOTAL
                                      ;NO. OF ARGUMENTS ON THIS
                                      ;CALL
          BYTE          'AGAIN'       ;GENERATES ONE BYTE OF
                                      ;MEMORY CONTAINING THE
                                      ;CURRENT PARAMETER
AGAIN     SET           AGAIN + 1     ;INCREMENT AGAIN AT
                                      ;ASSEMBLY TIME
          ENDR                        ;END OF REPEAT CONDITION
          BYTE          0DH           ;GENERATES A CARRIAGE
                                      ;RETURN
          ENDM                        ;END OF MACRO DEFINITION
```

Assume the following macro call appears within a main program.

```
LABEL     OPERATION     OPERAND        COMMENT
          CONDR         25,26,27       ;INVOKES MACRO CONDR AND
                                       ;SUBSTITUTES THE ARGUMENTS
                                       ;25, 26, AND 27 FOR THE FIRST
                                       ;THREE PARAMETERS
```

This macro call generates the following macro expansion and substitutes the arguments 25, 26, and 27 for the parameter labeled 'AGAIN'. The substitutions occur for as many times as there are arguments specified in the macro call, as defined by the '#' character. In this case, there are three arguments specified and the '#' character is replaced by 3.

```
                  0001    AGAIN   SET     1
                  FFFF    +       REPEAT  AGAIN<=00003
          0000    19      +       BYTE    25
                  0002    AGAIN   SET     AGAIN+1
                                  ENDR
                  FFFF    +       REPEAT  AGAIN<=00003
          0001    1A      +       BYTE    26
                  0003    AGAIN   SET     AGAIN+1
                                  ENDR
                  FFFF    +       REPEAT  AGAIN<=00003
          0002    1B      +       BYTE    27
                  0004    AGAIN   SET     AGAIN+1
                                  ENDR
          0003    0D              BYTE    0DH
                                  ENDM
00005     0004            END
```

MACRO EXPANSION SUMMARY

The lines of code within the macro definition block are not assembled with the rest of the program, but are saved until macro expansion time. Blank lines or comment lines are exceptions to this rule since they are not saved for expansion. The macro definition block, therefore, does not generate object code upon assembly. When the macro name appears within the operation field of the main program during assembly, the body of the macro is inserted and assembled within the main program.

Prior to the assembly of each line in the macro definition block, the assembler scans for the presence of the single quote character. An argument defined in the macro call then replaces the parameter within the single quote characters. After substitution, the scan continues from the first character following the replaced text until the end of the current line. The line is inserted into the user program. The assembler then generates object code and processes the line. The assembler continues to obtain lines from the macro definition block in this manner until an ENDM or EXITM statement is encountered. At that time, expansion continues with the statement following the macro call.

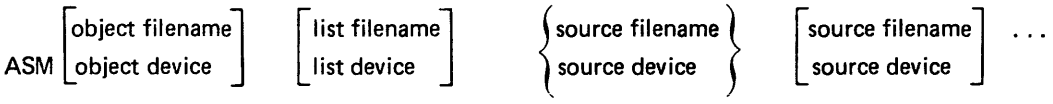
Section 6

ASSEMBLER OPERATING PROCEDURES

INTRODUCTION

This section describes the syntax required for the Tektronix Assembler to translate source code into executable binary object code.

SYNTAX



PURPOSE

The ASM command invokes the assembler when the 8002 μProcessor Lab is under TEKDOS control.

EXPLANATION

The optional object device or filename parameter causes the assembler to output the binary object module to the specified disc file or device. The optional list filename or device parameter causes the assembler to output a listing to the specified device or disc file. The source filename or device parameter specifies the source module to be translated.

All parameters within the ASM command line must be separated either by spaces or by commas. The object filename or device parameter is optional and, if omitted, must be replaced by two commas in the following manner. In this case an object file is not generated.

```
ASM, ,LIST SOURCE
```

The list filename or device parameter is also optional and, if omitted, must be replaced by two commas in the following manner. In this case an assembled listing is not generated.

```
ASM OBJECT, ,SOURCE
```

If the object and list filenames or devices are both omitted, they must be replaced by three commas in the following manner.

```
ASM, , ,SOURCE
```

If the object and list files are intended to reside on a disc other than the system disc, the appropriate disc drive number must follow the slash character (/) in the following manner.

```
ASM OBJECT/1 LIST/1 SOURCE
```

At least one source filename or device must be specified in the ASM command line. More than one source filename or device is acceptable if the ASM command and its parameters do not exceed one line. If the source file is stored on a disc other than the system disc, the appropriate disc drive number must be specified after the / character in the following manner.

```
ASM OBJECT LIST SOURCE/1
```

If the specified source module is a device, the assembler source code must be entered twice; once for each assembler pass. In addition, if the source module is the console input device (CONI), care should be taken to ensure that the source code is entered exactly the same for both assembler passes.

ASSEMBLY COMPLETION

After assembly completion, each line containing an error is displayed along with an error code describing the nature of the error. Refer to Appendix F for a list of all error codes, messages, and their explanations. Below all error displays, two lines appear on the output device showing the number of source lines, the number of assembled lines, the number of available bytes, and the number of errors and undefined symbols. If an irrecoverable assembly error occurs, the program aborts and a message indicates the error in the followng form:

```
FATAL ERROR, ASSEMBLY ABORTED AT LINE XXXX
```

The TEKDOS prompt character (>) appears after all assembler messages have been displayed indicating assembly completion.

If an object filename or device parameter has been specified in the ASM command line, the translated program is stored as relocatable binary object code. A correctly assembled object file may be linked, and then executed or debugged.

If a list filename or device parameter has been specified in the ASM command line, the assembled listing is output to a device or disc file.

Section 7

ASSEMBLER LISTING FORMAT

INTRODUCTION

The assembler listing is composed of two parts:

1) the source program assembler listing with the object code generated for each instruction; and

2) a table of all symbols used in the program.

THE ASSEMBLER LISTING

The assembler listing is composed of headings, lines of source code listing information, and error responses relating to any assembling errors.

Headings

Each page of the assembler listing contains a heading. The heading includes the assembler version on the left side of the page, and the page number on the right side of the page, as shown below:

```
TEKTRONIX 9900 ASM Vx.x                                    PAGE X
```

If the TITLE directive is used, a 30-character string expression may be inserted at the top of each listing page for program identification. The character string specified as the TITLE operand is printed on the first character line between the assembler version number and the page number, shown as follows.

```
TEKTRONIX 9900 ASM Vx.x  THIS IS THE PROGRAM TITLE         PAGE X
```

If the STITLE directive is used, a 72-character string expression may be inserted on the second line of each listing page for program identification. The character string specified as the STITLE operand is printed between the page heading and the first source code line. A blank line is automatically inserted between the string and the beginning of the source code. A program identification heading created with the STITLE directive appears below:

```
TEKTRONIX 9900 ASM Vx.x                                    PAGE X
THIS LINE DEMONSTRATES STITLE USAGE
(blank line)
.
.  (source code)
.
```

The Listing Line

The heading is followed by a blank line and the listing information. Each source program line is translated and output in the following sequence:

1) a line number,
2) the memory location of the instruction or data,
3) the translated object code,
4) a relocation indicator if relocation occurs on the line,
5) a substitution indicator if substitution occurs on the line, and
6) the original source line.

The listing line may be 72 or 132 characters wide, dependent upon whether the TRM option for the LIST and NOLIST directives is active. The first listing line field is a five-character decimal line number. Line numbers are not listed for macro expansion lines. The second listing field is a four-character hexadecimal location counter. This field may also represent a symbol value for an EQU directive. Both the line number and the location counter are right justified with leading zeros when necessary, and are separated from each other by one space.

The object field follows the location counter field, and the fields are separated by one space. The object code is left justified and may be a maximum of twelve hexadecimal characters wide. If an instruction generates more than twelve hexadecimal characters, all additional object code is listed on subsequent lines.

If relocation occurs in a line, the greater-than character (>) follows the object field. Actual relocation is performed at link time.

If a substitution occurs in a line, the plus character (+) follows the relocation indicator or the object field. All substitutions occur before the line is listed. The example that follows shows the plus sign preceding a line where a substitution occurs.

```
0001 0000 030502     + BYTE 3,5,2          ;ALLOCATE ONE BYTE OF
                                           ;MEMORY FOR EACH OF THE
                                           ;CONSTANT VALUES 3 AND 5,
                                           ;AND FOR THE VALUE DEFINED
                                           ;TO SUBSTITUTE FOR '1' (IN
                                           ;THIS CASE THE VALUE IS 2)
```

The source code follows the relocation or substitution indicators or the object code field, and the fields are separated by one space. If the TRM option is ON when entered with the LIST directive, 47 spaces remain in the listing line for the source code. Any source code exceeding the 47-character limit is truncated. If the TRM option is OFF, whether by default or when entered with the NOLIST directive, 103 characters remain in the listing for the source code. Any source code exceeding the 103-character limit is truncated.

Any non-printing character, other than the space, tab, or carriage return characters, is represented by a question mark (?) in the listing. The assembler translates the character replaced by the ? to the original character form.

To summarize, the listing line appears as follows.

```
XXXX  LLLL  D D D D D D D D > + SSSSS . . . . .
```

Each field is represented as follows:

X = Line number, right justified

L = Memory location (or EQU statement symbol value)

D = Object code

> = Relocation indicator (relocation is performed at link time)

\+ = Substitution indicator (substitution has occurred before listing)

S = Source line

Error Response

If an error occurs in an instruction, the line containing the error is followed by an error response. This is also true when the instruction generates more than one line of object code. The error response takes the following form:

```
***** ERROR code
```

The "code" in the above error response is replaced by a three-digit number indicating the type of error detected. For a description of all error codes and their corresponding messages, refer to Appendix F. If the error response precedes an additional message, "FATAL ERROR; ASSEMBLY ABORTED AT LINE XXXX", the severity of the error is such that the Assembler cannot continue execution.

THE SYMBOL TABLE

The symbol table follows the listing, indicating all symbols used in the source module and the values these symbols represent. The symbol table also categorizes all symbols according to their type or base, for ease in referencing. The structure of the symbol table follows a three-part format: a heading, symbols and their values (categorized by type or base), and two lines providing statistical program assembly information.

Each symbol table page contains a heading following the format shown below:

TEKTRONIX 9900 ASM Vx.x SYMBOL TABLE LISTING PAGE X

Below the heading, symbols and their corresponding hexadecimal values appear in categories according to their type or base. Headings precede each category describing the group of symbols in each category. The possible symbol headings are as follows:

STRING AND MACROS	All string and macro symbols are listed under this category.
SCALARS	All symbols having scalar values and all undefined symbols are listed under this category. Additionally, all 9900 Microprocessor registers (RØ through R15) and their values are listed under this category.
name SECTION characteristic (length)	All symbols based to the named Linker section are listed. If specified, the section characteristic indicates that the section is based to the actual address specified by the ORG directive at assembly time (ABSOLUTE). Refer to the discussion on Section Definition Directives in Section 4. If no characteristic is listed, the section is byte relocatable. The length of the named section is specified in bytes.

name COMMON characteristic (length)	Same as SECTION category, except that more than one common section with the same name is valid at link time.
name RESERVE characteristic (length)	Same as SECTION category, except that all sections with the specified name are combined into a single section at link time.
name UNBOUND GLOBAL	An unbound global is a symbol declared in a global statement, having no value in this assembly. The named unbound global must be defined in other assemblies or at link time. If an unbound global is used to assign a value to a symbol in this assembly, that symbol is listed under the UNBOUND GLOBAL category in the symbol table listing.

Columns containing symbols and their corresponding hexadecimal values are listed alphabetically under each category. When a symbol has fewer than eight characters, dashes and spaces (– – –) serve as padding between a symbol and its value. The value field contains four hexadecimal characters and is right justified, with leading zeros where necessary. The value field for undefined symbols appears as a series of asterisks (****). Each value is followed by several spaces and the next symbol. A typical symbol table listing line might appear as follows:

SYM1 ––– Ø1Ø1 SYMB2 — ØØ25 SYMB3 –– ØØ22 SYMBOL4 **** SYMBOL5 Ø121

The number values for string and macro symbols indicate the number of bytes used by the symbol for text storage. The number values for SET symbols indicate the last values assigned to the symbols. The number values for GLOBAL and ENDOF symbols represent the addresses prior to relocation.

Symbol indicators may appear after the symbol values. An indicator also appears if a high or low truncation occurs at link time. The symbol indicators are summarized as follows:

S – String symbol

M – Macro symbol

V – SET symbol

G – Global symbol

H – High truncation indicator (truncation will occur at link time)

L – Low truncation indicator (truncation will occur at link time)

E – ENDOF symbol (value will be adjusted at link time)

All symbols without indicators are EQU symbols. The number values for these symbols indicate their values during assembly.

If the TRM option is specified with the NOLIST directive, or is otherwise OFF due to default, the symbol table listing is five columns wide. If the TRM option is specified with the LIST directive, causing the option to be ON, the symbol table listing is three columns wide.

Two lines appear below the symbol table display providing statistical information about the current assembly. The first line shows the number of source lines, the number of assembled lines, and the number of available bytes. The number of available bytes indicates the amount of space available for further data manipulation or symbol storage within the assembler. The second statistical line indicates the number of errors and undefined symbols, if any.

A sample assembler and symbol table listing is shown in Fig. 7-1.

```
TEKTRONIX 9900 ASM V3.Ø     THIS IS THE TITLE                          PAGE 1
THIS LINE IS THE STITLE OF MY PROGRAM

ØØØØ3                              STRING   S1(8Ø)      ;DEFINE STRING VARIABLE S1 WITH
                                                        ;80-CHARACTER MAXIMUM
ØØØØ4              ØØØ3  L1        EQU      3          ;DEFINE CONSTANT SYMBOL L1 TO
***** ERROR   ØØ3                                       ;EQUAL 3

ØØØØ5              ØØØ4  L2        SET      4          ;DEFINE VARIABLE SYMBOL L2 TO
                                                        ;EQUAL 4
ØØØØ6              1ØØ>            ORG      1ØØH       ;STARTS OBJECT CODE OF NEXT
                                                        ;INSTRUCTION AT 1ØØH
ØØØØ7  1ØØ   CØ81      L1         MOV      R1,R2      ;LOAD THE CONTENTS OF REGISTER R1
***** ERROR   ØØ2                                       ;INTO REGISTER R2. MULTIPLY-DEFINED
                                                        ;SYMBOL, L1.

ØØØØ8                              END                 ;END OF PROGRAM
.
.
.
TEKTRONIX 9900 ASM V3.Ø     SYMBOL TABLE LISTING                       PAGE 2

STRINGS AND MACROS

      S1 - - - - - ØØ5Ø  S

SCALARS

      L2  --- ØØØ4V   RØ  --- ØØØ6   R1  --- ØØØ1
      R2  --- ØØØ2    R3  --- ØØØ3   R4  --- ØØØ4
      R5  --- ØØØ5    R6  --- ØØØ6   R7  --- ØØØ7
      R8  --- ØØØ8    R9  --- ØØØ9   R10 --- ØØØA
      R11 --- ØØØ8    R12 --- ØØØC   R13 --- ØØØD
      R14 --- ØØØE    R15 --- ØØØF

% (default) SECTION (Ø1Ø1)

      L1 - - - - - Ø1ØØ

15 SOURCE LINES       15 ASSEMBLED LINES       1ØØØ BYTES AVAILABLE
2 ERRORS
```

Fig. 7-1. Sample Assembler and Symbol Table Listing.

Section 8

ASSEMBLER OBJECT MODULE FORMAT

INTRODUCTION

The TEKTRONIX Assembler object module output can be stored on flexible disc in binary code. The binary object code may then be linked or loaded into program memory for execution and debugging. If a module contains more than one section or references global symbols declared in other modules, it must be linked before loading its object code into program memory.

PROGRAM LOADING AND EXECUTION

The TEKDOS command, LOAD, is used to load an assembled binary object file or linked load module into program memory. The TEKDOS command, GO, may then be entered to begin program execution or debugging. The following descriptions outline LOAD and GO command usage. For further details describing binary object code execution procedures, refer to the EMULATOR ENVIRONMENT section in the 8002 μProcessor Lab System User's Manual.

SYNTAX

LOAD {file name [/disc drive]} [file name [/disc drive]] ...

PURPOSE

The LOAD command program loads Assembler and Linker object files into program memory.

EXPLANATION

The specified file name is loaded into program memory with the LOAD command. The file must have been previously created by the Assembler or the Linker. Assembler object files containing relocatable sections or references to global symbols may execute incorrectly if not linked before loading.

The named file is loaded into program memory starting at the location specified in the source code.

Possible *DOS* error responses for the LOAD command are as follows:

6 – Device read error
14 – Invalid input device
48 – Load file not found
49 – Load file assign failure
51 – Invalid load request

SYNTAX

GO [address]

PURPOSE

The GO command causes execution control to be passed to the emulator processor.

EXPLANATION

The GO command causes execution control to be passed to the emulator processor with execution to begin at the specified address. When the address is not specified, execution begins at the start address of a previously loaded module, or continues from the last stopping point.

The GO command is a forced jump and will supersede a RESET command.

The possible *DOS* error response for the GO command is shown below:

37 – Invalid GO address

Section 9

THE LINKER

INTRODUCTION

The 8002 μProcessor Lab Assembler converts user-written program instructions into machine language modules, each module consisting of one or more sections. The Linker, a system utility program, merges the independently assembled sections into an 8002 load file.

The Assembler creates machine-language output files, which the Linker may then convert into a single binary file, suitable for loading into program memory by the LOAD command.

The object files output from the Assembler consist of Text Blocks, Relocation Blocks, and Global Symbol Directory Blocks. Text Blocks from an independently assembled program section consist of three distinct item types:

1. Constants and machine instructions whose values are independent of their position in memory;

2. Addresses or address constants whose values are relative to the starting location (base) of a section; and

3. Global references to other object modules whose values cannot be determined until all sections are assegned memory locations.

This information is in binary data form.

Relocation Blocks contain information necessary to update and relocate bytes of program text. Global Symbol Directory Blocks define global symbols and sections.

Each microprocessor supported by the 8002 μProcessor Lab has unique qualities. The Linker supports these unique qualities and permits the interchange of microprocessors within the 8002. The Linker's outward appearance and operational method remain the same, regardless which microprocessor is supported.

To prepare object modules for the 8002 Load program, the Linker performs three specific functions:

1) Allocates memory space for each section of the loadable program;
2) Establishes a reference table of global symbols; and
3) When necessary, relocates address-dependent locations to correspond to allocated space.

In addition, the Linker generates a listing that indicates where sections are allocated, and the values of all global symbols.

LINKER INVOCATION

Three methods of Linker invocation are available: simple invocation, interactive command invocation, and command file invocation. Simple invocation requires entry of filenames only; all other parameters are set to reasonable default values. This method is usually adequate for most linking situations.

For more precise control, the Linker can be invoked by an interactive command series using default or user-specified parameters. The user can specify section attributes and section location, define global symbols, and control the listing content.

Linker activation through command file invocation is accomplished by specifying a named file containing a Linker command series.

PROGRAM SECTIONS

A section is a collection of object code that has been assembled with the same location counter. An object module may consist of several sections. These sections are treated separately by the Linker and each section is independently relocatable. No limit is placed on the number of sections per link, but no more than 255 sections or globals may exist in any one object module.

SECTION ATTRIBUTES

A section has four attributes that provide the Linker with information regarding memory allocation and where to link the section. These attributes are Name, Section Type, Size, and Relocation Type.

NAME

A section has an eight-character Name, assigned by the section directives, COMMON, RESERVE, and SECTION at assembly time. The Name must be a valid identifier. The section Name is entered into the Linker's symbol table and is a valid external symbol.

SECTION TYPE

A section may be either a SECTION, RESERVE, or COMMON. The specification is made through use of the SECTION, RESERVE, or COMMON directive at assembly time.

Each SECTION Name must be different. Multiple SECTIONS with the same name will be flagged as errors, and only the first one will be linked.

RESERVE sections with the same name are concatenated by the Linker. The length of a RESERVE section in a load module is the sum of all RESERVE sections with the same name.

COMMON sections with the same name are allocated the same space in memory. The length of the linked COMMON is that of the largest COMMON section.

SIZE

The size of each section in an object file is determined at assembly time. Section size is the number of program memory bytes that the section may occupy.

RELOCATION TYPE

A section may be Absolute (non-relocatable) or Byte Relocatable.

An Absolute section is not relocated by the Linker. Memory locations in an Absolute section where code has been generated, or locations that have been explicitly reserved by the assembler BLOCK directive, are not allocated to any relocatable section at link time. However, if two or more absolute sections have code at the same address, the contents of those memory locations after linking is undefined. These memory conflicts, if they occur, are noted on the Linker memory map.

A Byte Relocatable section can be placed anywhere in memory.

LINKER INVOCATION

Simple Invocation

SYNTAX

LINK [load file] [list file] {object1} [,object2] . . .

The "load file" represents the name to be assigned to the Linker-created load module. The "list file" represents the listing file name, and "object1", "object2" are input object files to be linked.

With simple invocation, all filename parameters must be entered on one line. No other parameter entries are permitted. If filenames for load file or list file are not entered, a null specification is assumed and the corresponding file is not generated. A map and error messages are output to the list file, and error messages are also logged to the console.

Interactive Command Invocation

SYNTAX

LINK (carriage return)

The Linker responds with the prompt character (an asterisk), to indicate that a Linker command will be accepted. Each command must be terminated by a carriage return. Commands will be accepted until an END command is received. An END command directs the Linker to discontinue command entry mode and to begin processing the object files.

Command File Invocation

SYNTAX

LINK { @filename }

Commands are read from filename until an end of file or an END command is encountered. End of file or END directs the Linker to discontinue command mode and begin processing object files. If errors have been generated, the Linker aborts with the message:
ERRORS IN INDIRECT FILE, LINK ABORTED.

Commands

The following commands may be used in interactive or command file invocation:

1. LOG

   Print messages to the console and log commands on the list file if one has been specified. All commands are echoed to the list file after log has been indicated.

2. NOLOG

   Don't log Linker messages on the console.

3. MAP

   Generate a memory map in the list file. A memory map lists module names, section names and attributes, global symbols defined within sections, and undefined global symbols. (See Linker Output description for further information.)

4. NOMAP

   Do not output a memory map.

5. LIST filename or device

   Generate a list file named "filename". See the listing file description for contents of the list file. "filename" is any valid TEKDOS file specification. A disc drive can be specified by appending the drive number to "filename" (filename/0, or filename/1). Instead of a filename, any valid output device may be designated.

6. LOAD filename

   Create load file. This command directs the Linker to generate a load file named "filename". The file will contain the executable output of the Linker and can be loaded using the LOAD command.

7. DEFINE symbol1 = value, . . .

   Define symbol. Symbol is the name of a global symbol; value is a hexadecimal number.

8. LINK object1, object2, . . .

   Link object files. This command directs the Linker to include the specified object files in the load file.

9. LOCATE section name [,memory location] [,BYTE]

   Allocate section declaration. Allows the user to locate a section and/or redefine its relocation type. Note that redefining the relocation type of a section may cause the linked code to execute differently than intended. Valid parameters for memory location are:

   BASE (starting address)
   or
   RANGE (starting address, ending address)

   where starting and ending addresses are hexadecimal numbers.

   When BASE is specified in the LOCATE command, the Linker places the section at the designated starting location. If RANGE is specified, the Linker places the section between the starting and ending locations. If there is not enough space within the specified range, the section will not be linked.

   A section specified as ABSOLUTE at assembly may be changed to BYTE relocatable with the LOCATE command.

   Valid relocation types are PAGE, INPAGE, and BYTE.

10. @filename

    Indicate indirect command file. This command directs the Linker to obtain subsequent commands from filename. Commands are read from the filename until an end of file or an END command is encountered. Indirect commands are echoed on the console as they are read. Nested indirect command files are illegal; a command file may not contain an "@filename" command.

11. TRANSFER symbol or value

    Specify load module transfer address. "Symbol" is a global symbol and "value" is a hexadecimal number with a leading character ranging from Ø through 9. This transfer value supersedes any transfer address encountered in linking object modules.

12. END

    End command entry mode. If no errors have been generated in command file invocation, this command will terminate command entry mode and initiate the processing of object files. If errors are detected, an appropriate message is issued and control is returned to the system monitor.

If an error is detected during command entry, a caret (^) is printed below the line to indicate the error location. A message defining the error is also printed. Following are examples of errors during interactive command entry. Throughout the examples, Linker generated characters have been underlined.

```
*LINK FILE1 FILE2 3FILE
                  ^
INVALID FILE NAME

*LIST LISTFILE
*DEFINE A==BB
         ^
SYNTAX ERROR


*DEFIN   SYMBOL =  3
 ^
ILLEGAL COMMAND

*LOG NO PARAMETERS NEEDED
     ^
EXTRANEOUS INFORMATION IGNORED
```

If these errors had been contained in a command file, and if the LOG command had been activated, the errors would have been logged to the listing.

MEMORY LOCATION

At link time the user may specify a relocatable section location, in the form of either a base address or an address range where the section may be placed. The default range for a relocatable section is the entire address space of the microprocessor. If the user elects not to specify a location for a section, the Linker will locate the section. An Absolute section cannot be moved at link time.

MEMORY ALLOCATION OF SECTIONS

The Linker allocates memory in the sequence shown here.

1. Absolute sections.

2. Based sections.
   Based means a program section starting location has been specified by a LOCATE command.

3. Ranged byte relocatable sections.
   Ranged means the user explicitly declared a RANGE (starting location, ending location) with the LOCATE command at link time.

4. Byte relocatable sections.

Absolute and based sections are linked even if conflicts occur. A conflict exists when two or more sections have bytes at the same address. Other section types are not linked if a conflict occurs. If any conflict occurs during allocation, a memory conflict is noted on the memory map. The content of memory in the conflicting area is undefined.

ENDREL

ENDREL is a pre-defined symbol whose value is assigned at link time. After memory is allocated, ENDREL is assigned the value of the first memory address available for use. This address is 1 greater than the highest address used by a non-based relocatable section. All relocatable sections are located below the value of ENDREL. Absolute sections, or sections relocated using the LOCATE command with a BASE specified, may or may not be located above the ENDREL address.

The user can override the default by assigning any other value to ENDREL. If ENDREL is neither defined nor referenced, no value is assigned.

LINKER OUTPUT

Listing File

The listing file may be output either to a flexible disc file or to the console, line printer, or other output device.

The following information may be included in a Linker output listing.

	Command Invocation	Simple Linker Invocation
Non-Fatal Errors and Messages	If specified	Yes
Map	If specified	Yes
Symbol List	Yes	Yes
Linker Statistics	Yes	Yes

Error Messages

An explanation of the Linker error messages is located at the end of this manual section.

MAP

A map consists of two parts:

1. A memory map — An ordered listing of the memory allocated to sections. The list starts with the lowest allocated address and proceeds to the highest allocated address space of linked sections.

2. A module map — A listing of modules linked into the load file. The map contains information concerning sections and global symbols defined in each module.

```
TEKTRONIX 9900 LINKER Vx.x      MEMORY MAP                    PAGE 1

0000 – 003F   TVALUES     SECTION ABSOLUTE
0040 – 007F   ABSTABLE    SECTION ABSOLUTE
0080 – 03D0   FOO         SECTION BYTE
03D2 – 13D2   MAIN        SECTION BYTE
13D4 – 1454   VECTORS     COMMON BYTE

  NO ERRORS          NO UNDEFINED SYMBOLS
  3 MODULES          6 SECTIONS
  TRANSFER ADDRESS UNDEFINED
```

Addresses are starred '*' if a conflict (an overlap) with another section occurred during allocation. Section type is either SECTION, COMMON, or RESERVE. Relocation type is ABSOLUTE.

The TRANSFER ADDRESS identifies program starting location. After loading the program in this example, the appropriate command would be "GO 0".

```
TEKTRONIX 9900 LINKER Vx.x      MODULE MAP                    PAGE 2

FILE: FILE 1
MODULE:  MAINPROG
    ABSTABLE     SECTION ABSOLUTE 0000–007F
    FOO          SECTION BYTE 0080–03D0
    MAKEREC      0280
    MAIN         SECTION BYTE 03D2–13D2
    FIXVALUE     0808  GETCHAR  0932

FILE: FILE 2
MODULE:  TRANSFOR
    TVALUES      SECTION ABSOLUTE 0000–003F
    VECTORS      COMMON BYTE 13D4–1454
    XVALUE       13D4  YVALUE  13F4  ZVALUE  1434

FILE: FILE 3
MODULE:  INPUT
    EMPTYABS     SECTION ABSOLUTE *EMPTY*
    VECTORS      COMMON BYTE 13D4–1454
    APRIME       13E4  YPRIME  1404
```

The module map lists linked modules. An alphabetical list of sections and entry points appears for each module. If no sections were linked in a module, an appropriate message so indicates. If no room for section is available, an appropriate message so indicates.

SYMBOL LIST

A symbol list is an alphabetical list of all global symbols (sections and symbols) and their assigned values. If a symbol is undefined, its value field is starred.

TEKTRONIX 9900 LINKER Vx.x SYMBOL LIST PAGE 3

ABSTABLE	0000	APRIME	13E4	EMPTYABS	0000	FIXVALUE	0808
FOO	0080	GETCHAR	0932	MAIN	03D2	MAKEREC	0280
TVALUES	0000	VECTORS	13D4	XVALUE	13D4	YPRIME	1404
YVALUE	13F4	ZVALUE	1434				

LINKER STATISTICS

The Linker Statistics include the number of errors, the number of undefined symbols, the number of sections, the number of modules, and the transfer address.

THE LOAD FILE

The primary output from Linker processing is the Load file. A Load file is a subset of the Linker input object files with all references and relocation resolved. It consists of a Module Block, a Global Symbol Directory Block, Relocation and Text Blocks, followed by an END Block. Load files are read into program memory with the LOAD command.

ERRORS AND ERROR MESSAGES

Three classes of errors can be generated during Linker execution:

WARNINGS (W)	A potential problem may exist but the linked program can probably be executed.
ERRORS (E)	Linked program probably will not execute properly.
FATAL ERRORS (F)	Errors directly affecting the Linker's execution. The Linker closes all open channels and returns control to TEKDOS.

All error classes cause an appropriate message to be output to the LOG and LIST file or device. A fatal error will be output to the console even if NOLOG was specified.

ERROR MESSAGES AND EXPLANATIONS

F. LINKER INTERNAL ERROR AT nnnn

An error occurred in the Linker. Try linking again. If this error persists, carefully document the incident and submit an LDP Software Performance Report to Tektronix.

E. NO ROOM IN RANGE nnnn-nnnn FOR SECTION NAME

The SECTION length is greater than available contiguous memory in range nnnn through nnnn of allocated section memory.

F. INVALID OBJECT CODE FORMAT FOR FILE NAME
LOCATION = nnnn

The information in file is not valid input object format. Ascertain that all files to be linked have been assembled. Location is the internal Linker address where the object file error was detected.

F. UNABLE TO ASSIGN file or device name

A file name specified as an Input Object File does not exist, or File/Device is unavailable.

F. MEMORY FULL

Linker memory is totally allocated and linking has been terminated. The total number of globals, sections, or object files must be reduced in order to link in the available memory.

W. TRANSFER ADDRESS UNDEFINED

No transfer address was specified to the Linker either through the transfer command or by specifying "END (expression)" during assembly. When no transfer address is specified, the Linker creates transfer address Ø.

W. TRANSFER ADDRESS MULTIPLY DEFINED IN MODULE name FILE name

The module has attempted to redefine the transfer address previously specified by a linked module or by the transfer command. The Linker uses the first encountered transfer address to generate a transfer address for the load module. If no transfer address is specified, a transfer of Ø is generated.

W. RELOCATION TYPE OF SECTION name MULTIPLY DEFINED IN MODULE name FILE name

An attempt was made to redefine the section relocation type (Byte or Absolute). This occurs when the LOCATE command defined a relocation type that differs from that specified at assembly time. The error also occurs when relocation attributes of a COMMON or RESERVE section differ between modules. The Linker uses the first encountered relocation attribute to define the section.

E. Symbol name MULTIPLY DEFINED IN MODULE name FILE name

An attempt was made to redefine a Global Symbol or SECTION. This occurs when two modules both define a Global of the same name or when two SECTIONS have the same name. SECTION names must be unique. In the event of multiply defined SECTIONS, the Linker will only include the first one in the Load Module.

E. TRUNCATION ERROR AT nnnn IN MODULE name FILE name

The relocated value computed for byte relocation is too large to fit into one byte.

E. UNRESOLVED REFERENCE AT nnnn MODULE name FILE name

A reference to an undefined global or section was specified at this point in the object code. This occurs when a global is used in one module but was never defined. The unresolved reference is zero filled in the load file.

W. MICROPROCESSOR REDEFINED FROM "microprocessor" IN MODULE name FILE name

The current input module has been generated for a different microprocessor than the previous object modules. Differences between microprocessor definitions may cause incompatabilities during linking (e.g., page length, alignment, etc.).

E. SECTION name EXCEEDS MAXIMUM SIZE

Section length is greater than the address space of the microprocessor. The section is not included in the load module. This error may occur when a RESERVE is too long. The maximum size for 9900 is 64k bytes.

W. IMPLICIT REORIGIN TO Ø IN SECTION name IN MODULE name FILE name

The Linker processed an object file where code in an absolute SECTION wrapped around from location FFFFH to Ø.

E. SECTION name CHANGED FROM PAGE TO BYTE RELOCATABLE
Either:

1) the section was declared to be page relocatable and the Linker doesn't support paging for that microprocessor; or,

2) there was insufficient room for a paged section in available memory. The Linker will attempt to allocate memory for the SECTION on a Byte Relocatable Boundary.

F. LIST FILE I/O ERROR # nn
LOAD FILE
CONSOLE
COMMAND FILE
OBJECT FILE

This error indicates that the Linker was unable to read to or write from the specified file or device. The error number corresponds to the SVC status byte.

COMMAND PROCESSING ERRORS

Extraneous Information Ignored

Extra characters are on a command line that only requires an instruction (e.g., LOG, NOLOG, MAP). The Linker performs the appropriate action for the command, ignoring extra characters on the line.

Illegal Command

The command was not recognized.

Syntax Error

Statement syntax is invalid. This occurs when a command is incorrectly formed. For example, unmatched parentheses are found in the LOCATE command, or an operand is missing after the equal sign in the DEFINE command.

Indirect File Depth Exceeded

A filename command was found during processing of an indirect command file. The command is ignored.

Invalid File Name

The file in a LIST, LOAD, or LINK command contains illegal file characters. The filename may not begin with a numeric character (0–9). One to eight characters from the following set are acceptable:

Alphabetic (A–Z), numeric (0–9), or special characters (" # & ' ( ) * ; = ?). An optional two-character disc drive indicator (/0 or /1) can follow the filename.

NOTE: Processing of the command line ceases when an invalid filename is encountered. All files up to the invalid filename, in the case of the LINK command, are added to the list of files to be linked.

Invalid Range Specified

The range (starting address through ending address) in the LOCATE command is invalid. The ending address must be greater than the starting address.

Section 10

9900 SERVICE CALLS

INTRODUCTION

A service call (SVC) allows the 9900 Emulator Processor to obtain peripheral service from the system processor during program execution. The SVC is an instruction sequence in the user program containing:

1) a 9900 compare word instruction (c), referring to the address of the emulator processor output port; and
2) a no-operation instruction, allowing time for the SVC to occur.

The SVC references the emulator processor output port address and cues the system processor that an I/O (input/output) function is to occur. The system processor then references a service request block pointer in the user program. The service request block (SRB) pointer in turn references a block of memory containing the actual service request I/O specifications. The I/O specification block is called the service request block (SRB). The SRB contains parameters such as:

1) the type of I/O to be performed,
2) the I/O device or file channel assignments, and
3) the size of buffers for data transfer.

With these parameters, the service call can then be executed within a defined SVC buffer area. A broader description of SVCs is given in the Service Call section of the 8002 μProcessor Lab System User's Manual.

SVC procedures specific to the 9900 Emulator Processor are described in this section. The specific procedures describe the way the 9900 SVC compare word (c) instruction refers to the SRB pointer word address. Table 10-1 shows the SRB pointer word address referred to by each 9900 SVC compare word instruction.

SVC	9900 SVC OUTPUT INSTRUCTION AND ADDRESS		SRB POINTER WORD ADDRESS
1	C	ØF1EEH,RØ	ØØ4ØH
2	C	ØF1ECH,RØ	ØØ42H
3	C	ØF1EAH,RØ	ØØ44H
4	C	ØF1E8H,RØ	ØØ46H
5	C	ØF1E6H,RØ	ØØ48H
6	C	ØF1E4H,RØ	ØØ4AH

Table 10-1. 9900 SVC Compare Word Instruction References

THE 9900 SVC COMPARE WORD OPERATION

The 9900 SVC compare word operation is initiated with the 9900 instruction, "C". The "C" instruction references the SRB pointer, which in turn references the appropriate SRB. The SRB then defines the peripheral I/O operations, and the buffer area where the I/O is to be performed. In the final step, peripheral I/O is performed within the defined buffer area.

An example of the 9900 SVC process follows. The program, named NEWPROG, uses an SVC that causes an ASCII line to be read into the SRB I/O buffer from the console input device. After the line is read into the buffer, the program halts. The comments to the right of each instruction explain the SVC execution sequence.

```
;PROGRAM TO READ ASCII DATA FROM THE CONSOLE INPUT DEVICE AND HALT

            SECTION     EXAMPLE,ABSOLUTE    ;DECLARES SECTION NAMED EXAMPLE
                                            ;TO BE NON-RELOCATABLE
CONFOR      ORG         Ø                   ;BEGINNING ADDRESS OF SVC LABELED
                                            ;CONFOR
;THE NEXT TWO LINES COMPOSE THE SVC
            C           ØF1EEH,RØ           ;SVC1
            NOP                             ;ALLOWS TIME FOR SVC TO OCCUR
            IDLE                            ;PROGRAM HALTS AFTER SVC IS
                                            ;COMPLETE
            ORG         Ø4ØH                ;BEGINNING ADDRESS OF SRB POINTER
;THE NEXT TWO LINES COMPOSE THE SRB POINTER
            WORD        CONSRB              ;RESULT IS WORD FOR SVC1, POINTS
                                            ;TO SRB
            ORG         11ØØH               ;BEGINNING ADDRESS OF SRB
;THE NEXT EIGHT LINES COMPOSE THE SRB
CONSRB      BYTE        1H                  ;READ ASCII AND WAIT
            BYTE        1H                  ;CHANNEL NUMBER 1
            BYTE        ØØ                  ;STATUS
            BYTE        ØØ                  ;SINGLE BYTE DATA
            BYTE        ØØ                  ;BYTE COUNT
            BYTE        CONIRD + 1          ;BUFFER LENGTH
            WORD        CONBUF              ;BUFFER POINTER
            ORG         2ØØH                ;BEGINNING OF BUFFER
CONIRD      EQU         8Ø                  ;MAX. INPUT LINE LENGTH LESS CR
;THE FOLLOWING LINE DEFINES THE SRB BUFFER AREA
CONBUF      BLOCK       CONIRD + 1;         ;DEFINES BUFFER FOR SVC
            END         CONFOR              ;SPECIFIES STARTING INSTRUCTION
                                            ;IN PROGRAM
```

The program is assembled and loaded as follows:

```
> ASM NEWOBJ NEWLIST NEWPROG
> LOAD NEWOBJ
```

Channel 1 is also assigned to the console input device. This assignment corresponds to the channel byte assignments in the preceding SRB.

```
> ASSIGN 1 CON!
```

Now the program is executed.

```
> GO Ø
```

The desired character string "STRING" is entered and read from the console input device as follows:

```
STRING
```

The ASCII characters S, T, R, I, N and G are now stored in the buffer.

The DUMP command may be used to display the hexadecimal contents of the buffer. The beginning address of the buffer was defined in the program as 2ØØH.

```
> DUMP 2ØØ

Ø2ØØ=53  54  52  49  4E  47  ØD  XX  XX  XX  XX  XX  XX  XX  XX  XX
      |   |   |   |   |   |
      S   T   R   I   N   G
```

(carriage return, followed by previous contents of program memory)

Section 11

9900 DEBUGGING

INTRODUCTION

Three debugging commands support the unique 9900 Emulator Processor architecture, and thus require special mention. These commands are summarized below, in the order in which they are presented in this section. For further debugging information, refer to the Debug System section of the 8002 μProcessor Lab System User's Manual.

COMMAND NAME	9900 DEBUGGING COMMAND SUMMARY
TRACE	Enables or disables program execution monitoring. When TRACE is enabled, program execution trace lines display the current instruction location, its hexadecimal representation, mnemonic, and operands. Trace lines also show the contents of the workspace pointer register (WP), status register (ST), and registers RØ through R15.
DSTAT	Display line shows the current status of the debugging session. The display line shows the emulator processor's next instruction address, all active breakpoints and their parameters, and the contents of the registers labeled WP, ST, and RØ through R15.
SET	Reassigns hexadecimal values to registers labeled WP, ST and RØ through R15.

SYNTAX

TRACE ALL [STEP] [[start address] {stop address}]
or
TRACE JMP [STEP] [[start address] {stop address}]
or
TRACE OFF

PURPOSE

The TRACE command enables or disables program execution monitoring.

EXPLANATION

When TRACE is enabled, program execution trace lines display the location of the current instruction, its hexadecimal representation, mnemonic, and operands. Trace lines also show the contents of the workspace pointer register (WP), the status register (ST), and all other registers as follows:

		R0	R1	R2	R3	R4	R5	R6	R7
ST	WP	R8	R9	RA	RB	RC	RD	RE	RF

The Trace Modes

The three trace modes are TRACE ALL, TRACE JMP, and TRACE OFF. When TRACE ALL or TRACE JMP is entered in the DEBUG mode, displayed trace lines allow program execution flow monitoring. TRACE ALL causes trace information for all instructions executed by the emulator processor to be displayed on the DEBUG display device.

TRACE JMP causes trace information to be displayed each time one of the following 9900 branch, return, or jump instructions causes trace information to be displayed:

B, BL, BLWP, RTWP, JMP, JLT, JLE, JEQ, JHE, JET,
JNE, JNC JOC, JNO, JL, JH, JOP, and XOP

If the STEP option is entered with either the TRACE ALL or TRACE JMP command, and a program is executed, control is returned to the DEBUG display device, allowing programmer intervention after each instruction's trace line is displayed.

When TRACE OFF is entered, all trace display is disabled.

The Trace Line

Each trace line resulting from TRACE ALL or TRACE JMP contains one program instruction and information pertinent to its execution. Displayed trace lines appear in the following format:

LOC	INSTRUCTION	ST	WP	REGISTERS							
Ø1ØØ	MOV 1234 * R5 +			ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ
	CD601234	CØØF	1ØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ

All trace line values are displayed in hexadecimal format. A description of the 9900 trace line follows:

LOC	The location of the last executed instruction.
INSTRUCTION	The first line shows the mnemonics of the last executed instruction. The second line is the hexadecimal representation of the instruction.
ST	The contents of the processor status register after the last instruction execution.
WP	The contents of the workspace pointer after the last instruction execution.
REGISTERS	The contents of the registers RØ through R15 corresponding to the above WP after the last instruction execution.

Debug Error Responses

Debug system error messages consist of the notation "* DEB *" and a number indicating the error type, as follows:

31	Parameter required
35	Invalid start address
36	Invalid end address
44	Invalid trace mode parameter

Trace Line Termination

In TRACE ALL or TRACE JMP mode, trace lines of all statements or all branch instructions, respectively, are continuously displayed during program execution. Tracing stops when one of the following occurs: (1) an end of job condition is reached, (2) a breakpoint suspends the display, (3) the space bar is pressed to suspend the display, (4) the IDLE instruction suspends the display, or (5) the ESC key is pressed to suspend program execution.

The ESC key may be pressed while the display has been suspended by a an IDLE instruction. To re-enter the TRACE mode, enter the following command:

```
GO  [address]
```

Execution then continues at the beginning of the IDLE instruction, if no other address is specified.

EXAMPLE

Suppose the following 9900 assembly language user program resides on your work disc:

```
LABEL   OPERATION   OPERAND      COMMENT
START   LWPI        Ø            ;SET WP
        LI          R9,14        ;SET START CLEAR ADDRESS
CONT    CLR         *R9+         ;CLEAR A WORD
        CI          R9,ØFBFEH    ;DONE?
        JNE         CONT         ;NO–CONTINUE
        IDLE                     ;YES–STOP
        END
```

The preceding program clears program memory from location 14 through FBFE. The program is assembled. Emulation mode Ø is assigned. The absolute binary object code is read into program memory with LOAD. Entering the DEBUG command as follows places the system in debug mode.

```
>DEBUG
```

The program may now be traced for errors in execution flow.

Suppose a continuous trace of all instructions in the program's execution sequence is desired. Enter the command sequence below. The appropriate trace lines follow.

```
>TRACE ALL
>GO Ø
```

LOC	INSTRUCTION	ST	WP	REGISTERS							
Ø000	LWPI Ø000			Ø2EØ	Ø000	Ø209	Ø014	Ø4F9	Ø289	FBFE	16FC
	Ø2300000	Ø000	Ø000	Ø34Ø	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
Ø004	LI R9, Ø014			Ø2EØ	Ø000	Ø209	Ø014	Ø4F9	Ø289	FBFE	16FC
	Ø2090014	Ø000	Ø000	Ø34Ø	Ø014	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
Ø008	CLR *R9+			Ø2EØ	Ø000	Ø209	Ø014	Ø4F9	Ø289	FBFE	16FC
	Ø4F9	Ø000	Ø000	Ø34Ø	Ø016	Ø000	FFFF	FFFF	FFFF	FFFF	FFFF
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.	.

Trace lines of all instructions are continuously displayed until a trace line termination condition is met.

SYNTAX

DSTAT

PURPOSE

The DSTAT command causes display of the current debugging session status.

EXPLANATION

The DSTAT command sends two display lines to the DEBUG display device permitting the debugging status observation. The display for the current line in a 9900 program takes the form below:

```
P=xxxx    BP=xxxx   W/R xxxx W/R  xxxx xxxx xxxx xxxx xxxx  xxxx xxxx xxxx
      ST=xxxx     WP=xxxx        xxxx xxxx xxxx xxxx xxxx  xxxx xxxx xxxx
```

All DSTAT display line values are in hexadecimal format. A description of the display line for a program written in 9900 assembly language follows.

P	The emulator processor's next instruction address.
BP	The two possible active breakpoints and the breakpoint parameters. If the R parameter is shown, a breakpoint is set to occur whenever an attempt is made to read from the specified breakpoint. If the W parameter is shown, a breakpoint is set to occur whenever an attempt is made to write from the specified breakpoint. If neither parameter is shown, a breakpoint is set to occur whenever an attempt is made to read from or write to the specified breakpoint.
ST	The value of the processor status word.
WP	The value of the processor workspace pointer.

The values of the registers RØ through R15 appear on the right-hand side of the DSTAT display entry. The values of RØ through R7 appear on the first line and the values of R8 through R15 appear on the second line.

EXAMPLES

Suppose breakpoints are set at addresses ØØØ8 and ØØØA in a 9900 program. Whenever an attempt is made to read (specified by "R") from either of these addresses, a breakpoint is set to occur. The following command lines set those breakpoints:

```
>BKPT ØØØ8 R
>BKPT ØØØA R
```

When the program is executed with the GO command, the first breakpoint occurs at address ØØØ8.

```
>GO
```

LOC	INSTRUCTION	ST	WP	REGISTERS							
ØØØ8	CL RØ			ØØØØ	1111	1111	1111	1111	1111	1111	1111
	Ø4CØ	CØØF	1ØØØ	1111	1111	1111	1111	1111	1111	1111	1111
	BREAK										

The second breakpoint occurs at address ØØØA:

```
>GO
```

ØØØA	LWPI 2ØØØ			ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ
	Ø2EØ2ØØØ	CØØF	2ØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ	ØØØØ
	BREAK										

```
>
```

A debug status line might now be useful to examine the current status of the debugging session.

```
>DSTAT

P=ØØØE  BP=ØØØ8        R ØØØA R    ØØØØ ØØØØ ØØØØ ØØØØ  ØØØØ ØØØØ ØØØØ ØØØØ
        ST=CØØF         WP=2ØØØ     ØØØØ ØØØØ ØØØØ ØØØØ  ØØØØ ØØØØ ØØØØ ØØØØ
```

The debug status line displays the emulator processor's next instruction address (ØØØE, since LWPI is a two-word instruction), the active breakpoints and their parameters (ØØØ8 R and ØØØA R), the workspace pointer (2ØØØ), the status register contents (CØØF) and the contents of the emulator processor registers, RØ through R15.

SYNTAX

SET {initial register} {first hex value} [second hex value] . . .

PURPOSE

To reassign hexadecimal values to the 9900 Emulator Processor registers, workspace pointer, or status register, enter the SET command line.

EXPLANATION

Values may be reassigned for a continuous series of one or more registers, beginning with the first register specified. This series should not exceed the available registers.

The 9900 Emulator Procesor may be reassigned in the following sequence:

RØ R1 . . . R15 WP ST

A description of the register sequence is outlined below:

RO	Register Ø
R1	Register 1
.	
.	
.	
R15	Register 15
WP	Workspace Pointer
ST	Status Register

Note when reassigning values to these registers, a two-byte hexadecimal value must be specified. When values under two bytes in length are specified, the high bytes of the registers are filled with zeros.

EXAMPLE

Suppose the register contents below are displayed by the DSTAT command:

```
P=Ø1Ø4                          ØØØØ  ØØØ1  ØØØ2  ØØØ3  ØØØ4  ØØØ5  ØØØ6  ØØØ7
      ST=ØØØF   WP=2ØØØ         ØØØ8  ØØØ9  ØØØA  ØØØB  ØØØC  ØØØD  ØØØE  ØØØF
```

To reassign zeros to registers R14 and R15, and change the workspace pointer to 2ØØ2, enter the following SET command line:

```
> SET R14  Ø  Ø  2ØØ2
```

Another look at the register contents with the DSTAT command shows the change.

```
>DSTAT

P=Ø1Ø4                          ØØØ1  ØØØ2  ØØØ3  ØØØ4  ØØØ5  ØØØ6  ØØØ7  ØØØ8
      ST=ØØØF   WP=2ØØ2         ØØØ9  ØØØA  ØØØB  ØØØC  ØØØD  ØØØØ  ØØØØ  FFFF
```

Since the new workspace pointer (WP) now points to location 2ØØ2 in memory, the register set is effectively changed. Register 1 becomes Register Ø, Register 2 becomes Register 1, and so on. Register 15 is filled with an unknown value (in this case, FFFF).

Section 12

PROTOTYPE CONTROL PROBE

INTRODUCTION

The prototype control probe links the prototype system to the emulator processor module. When this option is installed, the prototype microprocessor is replaced by the probe, permitting the prototype to be tested and debugged under 8002 μProcessor Lab control. Hardware debugging is accomplished through the emulator processor; the emulator software; and the probe, which substitutes for the microprocessor in the prototype. Programs written for execution by the microprocessor can be monitored completely, and emulation permits thorough prototype testing.

DESCRIPTION AND INSTALLATION

The prototype control probe consists of three connected parts: a 6-foot ground plane cable pair, driver/receiver board, and an 18-inch cable pair with a 64-pin plug. The complete assembly is shown in Fig. 12-1.

The 6-foot ground plane cable pair consists of two 40-conductor flat cables with ground, power, and signal lines. The free end of the cable pair connects to the emulator processor module by means of a cable termination card inserted at the top of the emulator board. The 9900 CPU is then moved to the prototype control probe module to minimize emulation delays.

Receivers for data, address, and circuit board control are located in the prototype control probe assembly. The module assembly provides signal integrity and minimizes loading on circuits connected to the microprocessor socket.

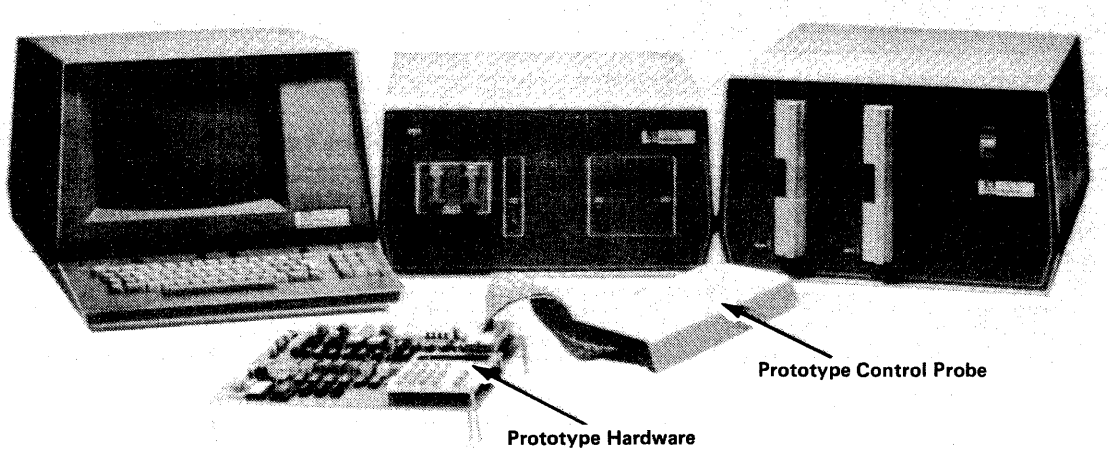


Fig. 12-1. 9900 Emulator Processor and Prototype Control Probe Assembly.

A 64-pin plug at the end of the 18-inch twisted-pair cables fits into the prototype microprocessor socket. Pin 1 on the plug must be mated to receptacle 1 on the socket. An indentation is located near pin 1 on the plug base to aid in pin identification. Refer to Fig. 12-2, demonstrating proper plug insertion.

CAUTION

If the plug is incorrectly inserted, damage to the prototype control probe will result. Fig. 12-2 illustrates the proper method for plug insertion.

If the plug is incorrectly inserted, the following parts may require replacement:

Within the Prototype Control Probe Driver/Receiver Board

DIP No.	Tektronix Part No.	Manufacturing No.
u1030	156-0956-00	74LS244
u1040	156-0956-00	74LS244
u1050	156-0480-00	74LS08
u1060	156-0383-00	74LS02
u2070	156-0480-00	74LS08
u4010	156-0480-00	74LS08
u4020	156-0928-00	74LS243
u4030	156-0928-00	74LS243
u4040	156-0928-00	74LS243
u4050	156-0928-00	74LS243
u4070	156-0383-00	74LS02

Emulator Processor Board

3AG 250 V 1A Fast Blow Fuse —— Tektronix Part No. 159-0022-00
3AG 250 V 3A Fast Blow Fuse —— Tektronix Part No. 159-0015-00

Buffer in Special Purpose Cable

Tektronix Part No. 156-0720-00 — Mfg. No. 74LS368 located in 64-pin probe

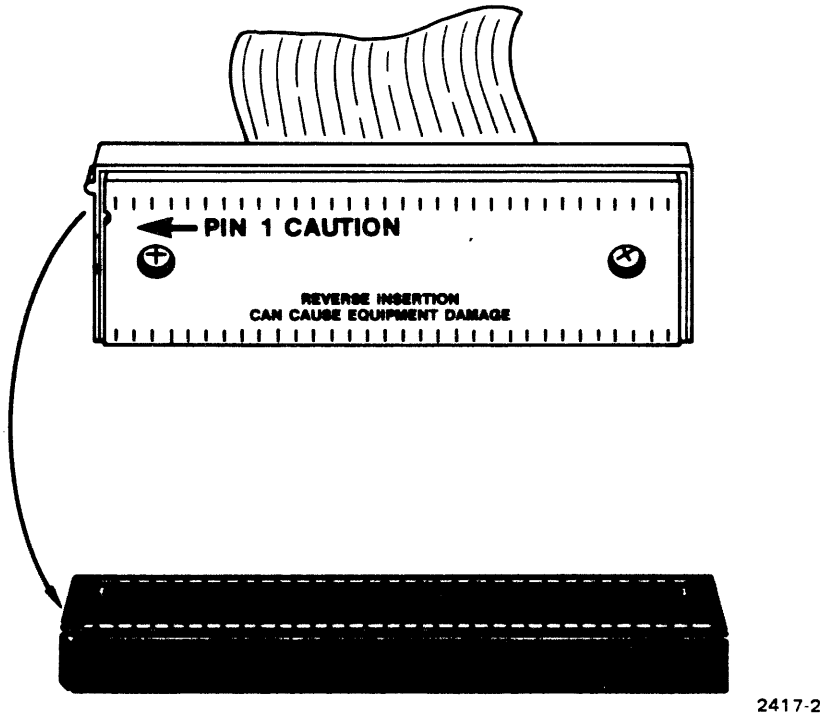


Fig. 12-2. Proper Plug Insertion.

When using the spring-plate protected 64-pin plug with a zero-insertion-force socket, place the 64-pin low-profile DIP socket (included) between the plug and the socket.

The prototype control probe, properly installed, is shown in Fig.12-3. If the pins on the plug are not shorted, the cable assembly can remain connected to the prototype hardware while the prototype control probe is not in use.

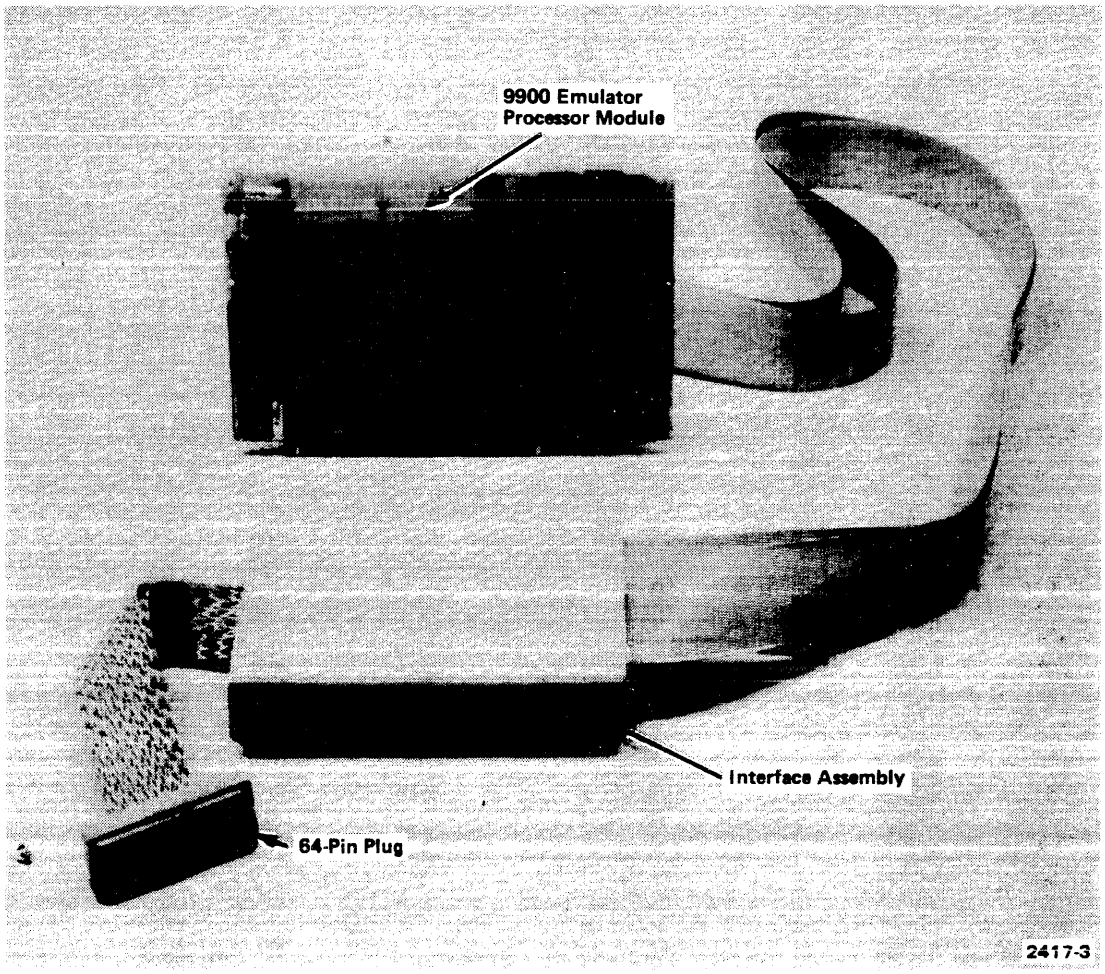


Fig. 12-3. Prototype Control Probe Connected to Prototype Hardware.

OPERATION

Once the prototype control probe is connected to the prototype hardware, the prototype hardware and software are exercised under TEKDOS control. Refer to the 8002 μProcessor Lab System User's Manual for details.

Appendix A

SOURCE MODULE CHARACTER SET

SYMBOLS	DEFINITION
A..Z	letters used in symbols; lower-case characters (other than in strings and comments) are interpreted as the corresponding upper-case characters
Ø . . . 9	numbers used in symbols and constants
$	used in symbols, and to represent assembler location counter contents
.	used in symbols
—	used in symbols
;	precedes a comment
, (comma)	delimiter for operand items
"	string delimiter
:	string concatenation operator
'	string substitution delimiter
#	total number of arguments passed to current macro expansion
[ ]	group macro code to be treated as a single argument
@	provides unique labels for each macro expansion
%	is replaced by name of current section or common in a macro expansion
*	binary arithmetic operation, multiplication
/	binary arithmetic operation, division
+	unary or binary arithmetic operator, addition
—	unary or binary arithmetic operator, subtraction
( )	override precedence of operators
\	unary logical operator, not
&	binary logical operator, and
!	binary logical operator, inclusive or
!!	binary logical operator, exclusive or
SPACE	field delimiter

SYMBOLS	DEFINITION
TAB	field delimiter
CARRIAGE RETURN	field and line delimiter
ALL ASCII CHARACTERS EXCEPT THE CARRIAGE RETURN CHARACTER	valid in string constants or in comments
∧ or ↑	allows following special character to have literal meaning
∧∧ or ↑↑	allows the second caret or up-arrow character to have literal meaning
=	relational operator, equal
< >	relational operator, not equal
>	relational operator, greater than
<	relational operator, less than
> =	relational operator, greater than or equal
< =	relational operator, less than or equal

Appendix B

ASSEMBLER DIRECTIVES

DIRECTIVE	OPERATION
ASCII	stores ASCII text in memory
BLOCK	reserves a specified number of bytes in memory
BYTE	allocates one byte of memory to each expression specified
COMMON	declares Linker section, assigns name, defines type to be common
ELSE	when expression is false, causes assembly of alternate source lines between ELSE and ENDIF directives
END	terminates source modules
ENDIF	signals corresponding IF block termination
ENDM	terminates a macro definition block
ENDR	signals end of each REPEAT cycle
EQU	permanently assigns a value to a symbol
EXITM	terminates expansion of current macro before encountering ENDM
GLOBAL	declares symbols to be global variables
IF	when expression is true, causes assembly of source lines between IF and ENDIF directives
INCLUDE	inserts text from specified file into the program
LIST	enables display of assembler listing features
MACRO	defines the name of a source code block used repeatedly within a program
NAME	declares name of an object module
NOLIST	disables display of assembler listing features
ORG	sets contents of location counter
PAGE	begins the next listing line on the following page
REPEAT	enables macro lines between REPEAT and ENDR directives to be assembled repeatedly

(Directives continued on next page)

DIRECTIVE	OPERATION
RESERVE	sets aside an area in memory
RESUME	continues definition of code for a given section
SECTION	declares Linker section, assigns name, defines parameters
SET	assigns or reassigns an expression value to a string or numeric variable symbol
SPACE	spaces downward a specified number of listing lines
STITLE	creates a text line on the second line of each listing page heading for program identification
STRING	declares symbol to be a string variable
TITLE	creates a text line at the top of each listing page heading for program identification
WARNING	generates specified warning message on the output device and in the listing
WORD	allocates two bytes of memory to each expression specified
WPNT	informs the Assembler of the location for the user's current workspace

ASSEMBLER DIRECTIVE SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ASCII	{string expression} [,string expression] . . .	[;charstring]
[symbol]	BLOCK	{expression}	[;charstring]
[symbol]	BYTE	{expression} [,expression] . . .	[;charstring]
[symbol]	COMMON	{symbol}[,ABSOLUTE]	[;charstring]
[symbol]	ELSE		
[symbol]	END	[expression]	[;charstring]
[symbol]	ENDIF		[;charstring]
[symbol]	ENDM		[;charstring]
[symbol]	ENDR		[;charstring]
{symbol}	EQU	{expression}	[;charstring]
[symbol]	EXITM		[;charstring]
[symbol]	GLOBAL	{symbol} [,symbol] . . .	[;charstring]
[symbol]	IF	{expression}	[;charstring]
[symbol]	INCLUDE	{string expression}	[;charstring]
[symbol]	LIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	MACRO	{symbol}	[;charstring]
[symbol]	NAME	{symbol}	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	ORG	{[/] expression}	[;charstring]
[symbol]	PAGE		[;charstring]
[symbol]	REPEAT	{expression1} [,expression2]	[;charstring]
[symbol]	RESERVE	{symbol, expression} [,ABSOLUTE]	[;charstring]
[symbol]	RESUME	[symbol]	[;charstring]
[symbol]	SECTION	{symbol} [,ABSOLUTE]	[;charstring]
{symbol}	SET	{expression}	[;charstring]
[symbol]	SPACE	[expression]	[;charstring]

(Directives continued on next page)

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	STITLE	{string expression}	[;charstring]
[symbol]	STRING	{{strvar1} [(lenexp1)]} [{,strvar2} [(lenexp2)]]...	[;charstring]
[symbol]	TITLE	{string expression}	[;charstring]
[symbol]	WARNING		[message]
[symbol]	WORD	{expression} [,expression] ...	[;charstring]
[symbol]	WPNT	{expression}	[;charstring]

Appendix C
SUMMARY OF 9900 INSTRUCTIONS

All 9900 instructions are summarized in this appendix. For a detailed description of the instruction set, consult a 9900 assembly language programming manual.

Each 9900 instruction statement consists of an operation code and up to two operands, depending upon the operation to be performed. An operation involving an implied operand consists of an operation code only. If an instruction involves data movement, the data flows from the first operand (origin) to the second operand (destination).

In the instruction summary that follows, the pre-defined symbols described in Section 2 are used in their correct context. Other operand notation is used as follows.

exp4	an expression representing a 4-bit XOP vector number, shift count, or CRU communications register unit bit count.
exp8	an expression representing a signed 8-bit CRU displacement address in the range −128 to 127.
exp16	an expression representing a 16-bit data or address constant.
r	one of the workspace registers, RØ to R15, or an expression that evaluates to a numeric register value.
rs or rd	represents one of the following (rs is the source and rd the destination):

r	one of the 16 workspace registers (Register Addressing Mode)
*r	the memory address contained in the workspace register specified by r (Register Indirect Addressing Mode)
*r+	the memory address contained in the workspace register specified by r. After this address is obtained, the contents of r are incremented by one (Register Indirect Auto Increment Addressing Mode)

If *r or *r+ is specified, the address in the register is treated as a word address, i.e., the least significant bit is ignored.

ms or md	represents one of the following (ms is the source address and md the destination address):

exp16	an address in memory (Memory Addressing Mode)
exp16(r)	the address in memory computed during program execution by adding the memory address specified by exp16 to the contents of the specified workspace register, r (Indexed Memory Addressing Mode)

This address is always a word (even-numbered) address. If an odd-numbered address is specified, the least-significant bit is ignored.

rsB or rdB	same as rs or rd except, that if Register Addressing Mode is specified, rsB or rdB refers to the high byte of rs or rd respectively. If Register Indirect or Register Indirect Autoincrement Addressing Mode is specified, the address contained in rs or rd is treated as a byte (odd or even) address.
msB or mdB	same as ms or md except the address is treated as a byte (odd or even) address.
rsH or rdH	same as rs or rd except only the high byte of rs or rd is used (the most significant byte).
rsL or rdL	same as rs or rd except only the low byte of rs or rs is used (the least significant byte).
msH or mdH	same as ms or md, except only the high byte of the contents of ms or md is used (the most significant byte).
msL or mdL	same as ms or md, except only the low byte of the contents of ms or md is used (the least significant byte).
(r, r + 1)	one of the workspace registers and the next higher numbered registers.
$R\emptyset_{\emptyset..3}$	bits Ø through 3 of RØ where bit Ø is the LSB.
$R12_{..12}$	bits 1 through 12 of R12 where bit Ø is the LSB.
rexp	a 16-bit memory address within the range −126 to 129 bytes from the current instruction. This address is translated to a PC-relative address, rad, in the object module.
rad	an 8-bit PC-relative address in two's complement form within the range −128 to 127. (This offset is relative to the address in the PC, the next instruction). rad is computed from rexp by the Assembler.
PC	the 16-bit program counter (points to the next instruction).
WP	the 16-bit workspace pointer register (points to the start of the workspace register area in memory).
LSB	least significant bit.
MSB	most significant bit.

I	the 4-bit interrupt control mask specifying the interrupt levels enabled.
ST	the 16-bit status register containing the condition codes and the interrupt control mask as follows:

15	14	13	12	11	10	9	8-4	3-Ø
L	A	Z	CY	V	P	EX		I

the meaning of the condition codes follows:

L	logical greater than status bit
A	arithmetic greater than status bit
Z	zero status bit
CY	carry-borrow status bit
V	overflow status bit
P	parity status bit
EX	extended operation status bit

the meaning of the condition code symbols follows:

u	status bit unaffected by instruction result
x	status bit set or reset depending on instruction result
z	the status bit may or may not be affected dependent on which instruction is executed by the XOP instruction. The XOP instruction does not affect the status bit.

AB_n	n^{th} bit of the address bus where Ø is the least significant byte.
←	indicates "is transferred to".
←→	indicates "is exchanged with".
+	addition operator.
–	subtraction operator.
*	multiplication operator.
÷	division operator.

| |	absolute value of expression between two bars.
QUO [ ]	quotient of the division specified in the brackets.
REM [ ]	remainder of the division specified in the brackets.
\	logical NOT operator.
&	logical AND operator.
!	logical inclusive OR operator.
!!	logical exclusive OR operator.
( )	refers to contents of address, register or flag.
(( ))	refers to the contents of allocation whose address is contained in the specified register (indirect addressing).
Memory Access	the total number of memory accesses needed to execute the instruction.
Machine Cycles	the number of system clock cycles needed to execute the instruction.
rsMemAcc	the number of memory accesses needed for the instruction located at the address contained in rs.
msMemAcc	the number of memory accesses needed for the instruction located at address ms.

NOTE

All 9900 instructions require additional memory accesses and machine cycles when Register Indirect, Register Indirect Autoincrement, or Indexed Memory Addressing modes are used. Refer to Table A for additional memory accesses and machine cycles for word instructions; refer to Table B for byte instructions.

Table A	if rs or rd = *r, add 1 to Mem Acc, and 4 to machine cycles if rs or rd = *r+, add 2 to Mem Acc, and 8 to machine cycles if ms or md = exp16(r), add 1 to Mem Acc
Table B	if rs or rd = *r, add 1 to Mem Acc, and 4 to machine cycles if rs or rd = *r+, add 2 to Mem Acc, and 6 to machine cycles if ms or md = exp16(r), add 1 to Mem Acc

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operand	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
DATA TRANSFER INSTRUCTIONS												
2	3	12	LI	r,exp16	(r)←exp16	x	x	x	u	u	u	u
2	2	10	LWPI	exp16	(WP)←exp16	u	u	u	u	u	u	u
1	4	14	MOV	rs,rd	(rd)←(rs)	x	x	x	u	u	u	u
2	5	22	MOV	rs,md	(md)←(rs)	x	x	x	u	u	u	u
2	5	22	MOV	ms,rd	(rd)←(ms)	x	x	x	u	u	u	u
3	6	30	MOV	ms,md	(md)(ms)	x	x	x	u	u	u	u
1	4	14	MOVB	rs,rd	(rdB)←(rsB)	x	x	x	u	u	x	u
2	5	22	MOVB	rs,md	(mdB)←(rsB)	x	x	x	u	u	x	u
2	5	22	MOVB	ms,rd	(rdB)←(msB)	x	x	x	u	u	x	u
3	6	30	MOVB	ms,md	(mdB)←(msB)	x	x	x	u	u	x	u
1	2	8	STST	r	(r)←(ST)	u	u	u	u	u	u	
1	2	8	STWP	r	(r)←(WP)	u	u	u	u	u	u	u
1	3	10	SWPB	rs	(rsH)←→(rsL)	u	u	u	u	u	u	u
2	4	18	SWPB	ms	(msH)←→(msL)	u	u	u	u	u	u	u
ARITHMETIC INSTRUCTIONS												
1	4	14	A	rs,rd	(rd)←(rd)+(rs)	x	x	x	x	x	u	u
2	5	22	A	rs,md	(md)←(md)+(rs)	x	x	x	x	x	u	u
2	5	22	A	ms,rd	(rd)←(rd)+(ms)	x	x	x	x	x	u	u
3	6	30	A	ms,md	(md)←(md)+(ms)	x	x	x	x	x	u	u
1	4	14	AB	rs,rd	(rdB)←(rdB)+(rsB)	x	x	x	x	x	x	u
2	5	22	AB	rs,md	(mdB)←(mdB)+(rsB)	x	x	x	x	x	x	u
2	5	22	AB	ms,rd	(rdB)←(rdB)+(msB)	x	x	x	x	x	x	u
3	6	30	AB	ms,md	(mdB)←(mdB)+(msB)	x	x	x	x	x	x	u
1	2	12	ABS	rs	(rs)← |(rs)|	x	x	x	x	x	u	u
2	3	20	ABS	ms	(ms)← |(ms)|	x	x	x	x	x	u	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
2	4	14	AI	r,exp16	(r)≷(r)+exp16	x	x	x	x	x	u	u
1	3	10	DEC	rs	(rs)←(rs) − 1	x	x	x	x	x	u	u
2	4	18	DEC	ms	(ms)←(ms) − 1	x	x	x	x	x	u	u
1	3	10	DECT	rs	(rs)←(rs) − 2	x	x	x	x	x	u	u
2	4	18	DECT	ms	(ms)←(ms) − 2	x	x	x	x	x	u	u
1	16	92-124	DIV	rs,r	If unsigned (rs) > unsigned (r), (r)←QUO[(r,r+1)÷(rs)] (r+1)←REM[(r,r+1)÷(rs)] Otherwise (v)←1, MemAcc = 3 and Cycles = 16	u	u	u	u	x	u	u
2	7	100-132	DIV	ms,r	If unsigned (ms) > unsigned (r), (r)←QUO[(r,r+1)÷(ms)] (r+1)←REM[(r,r+1)÷(ms)] Otherwise (v)←1, MemAcc = 4 and Cycles = 24	u	u	u	u	x	u	u
1	3	10	INC	rs	(rs)←(rs)+1	x	x	x	x	x	u	u
2	4	18	INC	ms	(ms)←(ms)+1	x	x	x	x	x	u	u
1	3	10	INCT	rs	(rs)←(rs)+2	x	x	x	x	x	u	u
2	4	18	INCT	ms	(ms)←(ms)+2	x	x	x	x	x	u	u
1	5	52	MPY	rs,r	(r,r+1)←unsigned(rs)*unsigned(r)	u	u	u	u	u	u	u
2	6	60	MPY	ms,r	(r,r+1)←unsigned(ms)*unsigned(r)	u	u	u	u	u	u	u
1	3	12	NEG	rs	(rs)← −(rs)	x	x	x	u	x	u	u
2	4	20	NEG	ms	(ms)← −(ms)	x	x	x	u	x	u	u
1	4	14	S	rs,rd	(rd)←(rd) −(rs)	x	x	x	x	x	u	u
2	5	22	S	rd,md	(md)←(md) −(rs)	x	x	x	x	x	u	u
2	5	22	S	ms,rd	(rd)←(rd)−(ms)	x	x	x	x	x	u	u
3	6	30	S	ms,md	(md)←(md)−(ms)	x	x	x	x	x	u	u
1	4	14	SB	rs,rd	(rdB)←(rdB)−(rsB)	x	x	x	x	x	u	u

@

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
2	5	22	SB	rs,md	(mdB)←(mdB)−(rsB)	x	x	x	x	x	u	u
2	5	22	SB	ms,rd	(rdB)←(rdB)−(msB)	x	x	x	x	x	u	u
3	6	30	SB	ms,md	(mdB)A(mdB)−(msB)	x	x	x	x	x	u	u
COMPARISON INSTRUCTIONS												
1	3	14	C	rs,rd	If unsigned (rs) > unsigned (rd); set (L)←1 If signed (rs) > signed (rd); set (A)←1 If (rs)=(rd); set Z←1	x	x	x	u	u	u	u
2	4	22	C	rs,md	If unsigned (rs) > unsigned (md); set (L)←1 If signed (rs) > signed (md); set (A)←1 If (rs)=(md); set (Z)←1	x	x	x	u	u	u	u
2	4	22	C	ms,rd	If unsigned (ms) > unsigned (rd); set (L)←1 If signed (ms) > signed (rd); set (A)←1 If (ms)=(rd); set (Z)←1	x	x	x	u	u	u	u
3	5	30	C	ms,md	If unsigned (ms) > unsigned (md); set (L)←1 If signed (ms) > signed (md); set (A)←1 If (ms)=(md) set (Z)←1	x	x	x	u	u	u	u
1	3	14	CB	rs,rd	If unsigned (rsB) > unsigned (rdB); set (L)←1 If signed (rsB) > signed (rdB); set (A)←1 If (rsB)=(rdB); set (Z)←1	x	x	x	u	u	x	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
2	4	22	CB	rs,md	If unsigned (rsB) > unsigned (mdB); set (L)←1 If signed (rsB) > signed (mdB); set (A)←1 If (rsB)=(mdB); set (Z)←1	x	x	x	u	u	x	u
2	4	22	CB	ms,rd	If unsigned (msB) > unsigned (rdB); set (L)←1 If signed (msB) > signed (rdB); set (A)←1 If(msB)=(rdB); set (Z)←1	x	x	x	u	u	x	u
3	5	30	CB	ms,md	If unsigned (msB) > unsigned (mdB); set (L)←1 If signed (msB) > signed (mdB); set (A)←1 If (msB)=(mdB); set (Z)←1	x	x	x	u	u	x	u
2	3	14	CI	r,exp16	If unsigned (r) > unsigned exp16; set (L)←1 If signed (r) > signed exp16; set (A)←1 If (r)= exp16; set (Z)←1	x	x	x	u	u	u	u
1	3	14	COC	rs,r	If (rs) & (r) = (rs); set (Z)←1	u	u	x	u	u	u	u
2	4	22	COC	ms,r	If (ms) & (r) = (ms); set (Z)←1	u	u	x	u	u	u	u
1	3	14	CZC	rs,r	If (rs) & (r) = Ø ; set (Z)←1	u	u	x	u	u	u	u
2	4	22	CZC	ms,r	If (ms) & (r) = Ø; set (Z)←1	u	u	x	u	u	u	u

©

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
LOGICAL INSTRUCTIONS												
2	4	14	ANDI	r,exp16	(r)←(r) & exp16	x	x	x	u	u	u	u
1	3	10	CLR	rs	(rs)←Ø	u	u	u	u	u	u	u
2	4	18	CLR	ms	(ms)←Ø	u	u	u	u	u	u	u
1	3	10	INV	rs	(rs)← \(rs)	x	x	x	u	u	u	u
2	4	18	INV	ms	(ms)← \(ms)	x	x	x	u	u	u	u
2	4	14	ORI	r,exp16	(r)←(r)!exp16	x	x	x	u	u	u	u
1	3	10	SETO	rs	(rs)←$FFFF_{16}$	u	u	u	u	u	u	u
2	4	18	SETO	ms	(ms)←$FFFF_{16}$	u	u	u	u	u	u	u
1	4	14	SOC	rs,rd	(rd)←(rd)!(rs)	x	x	x	u	u	u	u
2	5	22	SOC	rs,md	(md)←(md)!(rs)	x	x	x	u	u	u	u
2	5	22	SOC	ms,rd	(rd)←(rd)!(ms)	x	x	x	u	u	u	u
3	6	30	SOC	ms,md	(md)←(md)!(ms)	x	x	x	u	u	u	u
1	4	14	SOCB	rs,rd	(rdB)←(rdB)!(rsB)	x	x	x	u	u	x	u
2	5	22	SOCB	rs,md	(mdB)←(mdB)!(rsB)	x	x	x	u	u	x	u
2	5	22	SOCB	ms,rd	(rdB)←(rdB)!(msB)	x	x	x	u	u	x	u
3	6	30	SOCB	ms,md	(mdB)←(mdB)!(msB)	x	x	x	u	u	x	u
1	4	14	SZC	rs,rd	(rd)← \(rd) & (rs)	x	x	x	u	u	u	u
2	5	22	SZC	rs,md	(md)← \(md) & (rs)	x	x	x	u	u	u	u
2	5	22	SZC	ms,rd	(rd)← \(rd) & (ms)	x	x	x	u	u	u	u
3	6	30	SZC	ms,md	(md)← \(md) & (ms)	x	x	x	u	u	u	u
1	4	14	SZCB	rs,rd	(rdB)← \(rdB) & (rsB)	x	x	x	u	u	x	u
2	5	22	SZCB	rs,md	(mdB)← \(mdB) & (rsB)	x	x	x	u	u	x	u
2	5	22	SZCB	ms,rd	(rdB)← \(rdB) & (msB)	x	x	x	u	u	x	u
3	6	30	SZCB	ms,md	(mdB)← \(mdB) & (msB)	x	x	x	u	u	x	u
1	4	14	XOR	rs,r	(r)←(r)!!(rs)	x	x	x	u	u	u	u
2	5	22	XOR	ms,r	(r)←(r)!!(ms)	x	x	x	u	u	u	u

SHIFT AND ROTATE INSTRUCTIONS

Note: In all the following instructions, exp4 is an optional operand which is assumed to be zero when absent.

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L A Z CY V P EX
For exp4 ≠ Ø;			SLA	r,exp4	Shift (r) left. Fill vacated bits	x x x x x u u
1	3	12+2*exp4			with Ø.	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø					If exp4 ≠ Ø, shift (r) exp4 bits	
1	4	20+2*($R\emptyset_{\emptyset..3}$)			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø,	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø					shift (r) ($R\emptyset_{\emptyset..3}$) bits	
1	4	52			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø	
					shift (r) 16 bits.	
For exp4 ≠ Ø;			SRA	r,exp4	Shift (r) right. Extend sign bit	x x x x u u u
1	3	12+2*exp4			through vacated bits.	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø					If exp4 ≠ Ø, shift (r) exp4 bits	
1	4	20+2*($R\emptyset_{\emptyset..3}$)			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø,	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø					shift (r) ($R\emptyset_{\emptyset..3}$) bits	
1	4	52			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø	
					shift (r) 16 bits	
For exp4 ≠ Ø;			SRC	r,exp4	Shift (r) right. Bits shifted	x x x x u u u
1	3	12+2*exp4			out of ($r_\emptyset$) enter (r_{15})	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$)FØ					If exp4 ≠ Ø, rotate (r) exp4 bits	
1	4	20+2*($R\emptyset_{\emptyset..3}$)			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø,	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø;					(r) ($R\emptyset_{\emptyset..3}$) bits	
1	4	52			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø	
					shift (r) 16 bits	
For exp4 ≠ Ø;			SRL	r,exp4	Shift (r) right. Fill vacated	x x x x u u u
1	3	12+2*exp4			bits with Ø.	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø;					If exp4 ≠ Ø, shift (r) exp 4 bits	
1	4	20+2*($R\emptyset_{\emptyset..3}$)			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) ≠ Ø,	
For exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø					shift (r) ($R\emptyset_{\emptyset..3}$) bits	
1	4	52			If exp4 = Ø & ($R\emptyset_{\emptyset..3}$) = Ø	
					shift (r) 16 bits	

@

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
CONTROL TRANSFER INSTRUCTIONS												
1	2	8	B	rs	(PC)←rs	u	u	u	u	u	u	u
2	3	16	B	ms	(PC)←ms	u	u	u	u	u	u	u
1	3	12	BL	rs	(R11)←(PC) (PC)←rs	u	u	u	u	u	u	u
2	4	20	BL	ms	(R11)←(PC) (PC)←ms	u	u	u	u	u	u	u
1	6	26	BLWP	rs	(R13)←(WP) (R14)←(PC) (R15)←(ST) (WP)←(rs) (PC)←(rs+2)	u	u	u	u	u	u	u
2	7	34	BLWP	ms	(R13)←(WP) (R14)←(PC) (R15)←(ST) (WP)←(ms) (PC)←(ms+2)	u	u	u	u	u	u	u
1	1	10	JEQ	rexp	If (Z) = 1, (PC)←(PC)+rad Otherwise (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JGT	rexp	If (A) = 1, (PC)←(PC)+rad Otherwise (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JH	rexp	If (L) = 1 & (Z) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JHE	rexp	If (L) = 1 or (Z) = 1, (PC)←(PC)+rad Otherwise (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
1	1	10	JL	rexp	If (L) = Ø & (Z) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JLE	rexp	If (L) = Ø or (Z) = 1, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JLT	rexp	If (A) = Ø & (Z) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JMP	rexp	(PC)←(PC)+rad	u	u	u	u	u	u	u
1	1	10	JNC	rexp	If (CY) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JNE	rexp	If (Z) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JNO	rexp	If (V) = Ø, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JOC	rexp	If (CY) = 1, (PC)←(PC)+rad Otherwise (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	1	10	JOP	rexp	If (P) = 1, (PC)←(PC)+rad Otherwise, (PC)←(PC) and Machine Cycles = 8	u	u	u	u	u	u	u
1	3	12	RT		(PC)←(R11)	u	u	u	u	u	u	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
1	4	14	RTWP		(WP)←(R13) (PC)←(R14) (ST)←(R15)	x	x	x	x	x	x	x
2	2+rsMemAcc−1	8+rsCycles−4	X	rs	Execute instruction at rs. If this instruction requires one or two 16 bit operands, they must follow the X instruction	z	z	z	z	z	z	z
2	3+msMemAcc−1	16+msCycles−4	X	ms	Execute instruction at ms. If this instruction requires one or two 16 bit operands, they must follow the X instruction	z	z	z	z	z	z	z
1	8	36	XOP	rs,exp4	(R11)←(rs) (R13)←(WP) (R14)←(PC) (R15)←(ST) (WP)←(40_{16}+exp4*4) (PC)←(42_{16}+exp4*4)	u	u	u	u	u	u	x
2	9	44	XOP	ms,exp4	(R11)←(ms) (R13)←(WP) (R14)←(PC) (R15)←(ST) (WP)←(40_{16}+exp4*4) (PC)←(42_{16}+exp4*4)	u	u	u	u	u	u	x
COMMUNICATIONS REGISTER UNIT (CRU) INSTRUCTIONS												
1	3	20+2*exp4	LDCR	rs,exp4	Transfer bits serially from (rs) to CRU in LSB to MSB order. If exp4 ≠ 0, transfer the low order exp4 bits of (rs) to the CRU bit addresses, (starting at ($R12_{1..12}$))	x	x	x	u	u	x	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
					If 1≤exp4≤8, (rs) is a byte address If 9≤exp4≤15, (rs) is a word address and (P) is unaffected If exp4 = Ø, 16 bits of (rs) are transfered, (P) is unaffected and Machine Cycles = 52							
2	4	28+2*exp4	LDCR	ms,exp4	Transfer bits serially from (ms) to CRU in LSB to MSB order. If exp4 ≠ Ø, transfer exp4 of the low order bits of (ms) to the CRU bit addresses (starting at ($R12_{1-12}$)) If 1≤exp4≤8, (ms) is a byte address If 9≤exp4≤15, (ms) is a word address and (P) is unaffected If exp4 = Ø, 16 bits of (ms) are transferred, (P) is unaffected, and Machine Cycles = 52	x	x	x	u	u	x	u
1	2	12	SBO	exp8	Set a bit on the CRU (($R12_{1-12}$)+exp8)←1	u	u	u	u	u	u	u
1	2	12	SBZ	exp8	Reset a bit on the CRU (($R12_{1-12}$)+exp8)←Ø	u	u	u	u	u	u	u
1	4	58	STCR	rs,exp4	Transfer bits serially from the CRU to (rs) in LSB to MSB order. If exp4 ≠ Ø, transfer exp4 bits of the CRU (starting from ($R12_{1-12}$)) to (rs). Any unfilled bits in (rs), are set to Ø.	x	x	x	u	u	x	u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L A Z CY V P EX
					If 1≤exp4≤8, (rs) is a byte address and Machine Cycles = 44 (48 if exp4 = 8) If 9≤exp4≤15, (rs) is a word address and (P) is unaffected If exp4 = Ø, transfer 16 bits from CRU (starting at address ($R12_{1-12}$) to (rs), Machine Cycles = 6Ø and (P) is unaffected	
2	5	66	STCR	ms,exp4	Transfer bits serially from the CRU to (ms) in LSB to MSB order. If exp4 ≠ Ø, transfer exp4 bits of the CRU (starting from ($R12_{1-12}$)) to (ms) Any unfilled bits in (ms) are set to Ø. If 1≤exp4≤8, (ms) is a byte address and Machine Cycles = 50 (52 is exp4 = 8) If 9≤exp4≤8, (ms) is a word address and (P) is unaffected If exp4 = Ø, transfer 16 bits of the CRU (starting from ($R12_{1-12}$)) to (ms), Machine Cycles = 68 and (P) is unaffected.	x x x u u x u
1	2	12	TB	exp8	Test specified (RU bit (Z)←(($R12_{1-12}$)+exp8)	u u x u u u u

Object Module Words	Memory Access	Machine Cycles	Source Module Syntax Operation	Operands	Instruction Description	Condition Codes L	A	Z	CY	V	P	EX
INTERRUPT CONTROL INSTRUCTIONS												
1	1	12	CKOF		$(AB_{12})\leftarrow\emptyset$ $(AB_{13})\leftarrow 1$ $(AB_{14})\leftarrow 1$	u	u	u	u	u	u	u
1	1	12	CKON		$(AB_{12})\leftarrow 1$ $(AB_{13})\leftarrow\emptyset$ $(AB_{14})\leftarrow 1$	u	u	u	u	u	u	u
1	1	12	IDLE		Suspend instruction execution until an interrupt, LOAD or RESET occurs. $(AB_{12})\leftarrow\emptyset$ $(AB_{13})\leftarrow 1$ $(AB_{14})\leftarrow\emptyset$	u	u	u	u	u	u	u
2	2	16	LIMI	exp16	$(e)\leftarrow exp16_{\emptyset-3}$	u	u	u	u	u	u	u
1	1	12	LREX		$(AB_{12})\leftarrow 1$ $(AB_{13})\leftarrow 1$ $(AB_{14})\leftarrow 1$	u	u	u	u	u	u	u
1	1	10	NOP		No operation	u	u	u	u	u	u	u
1	1	12	RSET		$(I)\leftarrow\emptyset$ $(AB_{12})\leftarrow 1$ $(AB_{13})\leftarrow 1$ $(AB_{14})\leftarrow\emptyset$	u	u	u	u	u	u	u

Appendix D
SERVICE CALL FUNCTION CODES

CODE	FUNCTION
Ø1	Read ASCII and wait
Ø2	Write ASCII and wait
Ø3	Close device or file on channel
Ø4	Rewind file on channel
Ø5	Delete file on channel
Ø6	Rename file on channel
1Ø	Assign channel to device or channel
11	Get time (milliseconds)
12	Get overlay addresses
13	Get parameter (procedure parameter buffer)
14	Get device type
15	Get device status
16	Get last console input character
17	Load overlay
18	Execute overlay
19	Suspend execution
1A	Exit
1C	Get parameter (emulation parameter buffer)
1F	Abort
41	Read binary and wait
42	Write binary and wait
81	Read ASCII and proceed
82	Write ASCII and proceed
C1	Read binary and proceed
C2	Write binary and proceed

Appendix E

HEXADECIMAL CONVERSION TABLES

ASCII CODE CONVERSION TABLE

HEXADECIMAL

LEAST SIGNIFICANT CHARACTER	MOST SIGNIFICANT CHARACTER							
—	Ø	1	2	3	4	5	6	7
Ø	NUL	DLE	SP	Ø	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	≈
F	SI	US	/	?	O	—	o	DEL

EXAMPLES

W = 57
H = 48
a = 61
t = 74
@ = 40
NUL = ØØ
DEL = 7F

Decimal-Hexadecimal-Binary Equivalents 0-255$_{10}$

Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code
0	00	0000 0000	64	40	0100 0000	128	80	1000 0000	192	C0	1100 0000
1	01	0000 0001	65	41	0100 0001	129	81	1000 0001	193	C1	1100 0001
2	02	0000 0010	66	42	0100 0010	130	82	1000 0010	194	C2	1100 0010
3	03	0000 0011	67	43	0100 0011	131	83	1000 0011	195	C3	1100 0011
4	04	0000 0100	68	44	0100 0100	132	84	1000 0100	196	C4	1100 0100
5	05	0000 0101	69	45	0100 0101	133	85	1000 0101	197	C5	1100 0101
6	06	0000 0110	70	46	0100 0110	134	86	1000 0110	198	C6	1100 0110
7	07	0000 0111	71	47	0100 0111	135	87	1000 0111	199	C7	1100 0111
8	08	0000 1000	72	48	0100 1000	136	88	1000 1000	200	C8	1100 1000
9	09	0000 1001	73	49	0100 1001	137	89	1000 1001	201	C9	1100 1001
10	0A	0000 1010	74	4A	0100 1010	138	8A	1000 1010	202	CA	1100 1010
11	0B	0000 1011	75	4B	0100 1011	139	8B	1000 1011	203	CB	1100 1011
12	0C	0000 1100	76	4C	0100 1100	140	8C	1000 1100	204	CC	1100 1100
13	0D	0000 1101	77	4D	0100 1101	141	8D	1000 1101	205	CD	1100 1101
14	0E	0000 1110	78	4E	0100 1110	142	8E	1000 1110	206	CE	1100 1110
15	0F	0000 1111	79	4F	0100 1111	143	8F	1000 1111	207	CF	1100 1111
16	10	0001 0000	80	50	0101 0000	144	90	1001 0000	208	D0	1101 0000
17	11	0001 0001	81	51	0101 0001	145	91	1001 0001	209	D1	1101 0001
18	12	0001 0010	82	52	0101 0010	146	92	1001 0010	210	D2	1101 0010
19	13	0001 0011	83	53	0101 0011	147	93	1001 0011	211	D3	1101 0011
20	14	0001 0100	84	54	0101 0100	148	94	1001 0100	212	D4	1101 0100
21	15	0001 0101	85	55	0101 0101	149	95	1001 0101	213	D5	1101 0101
22	16	0001 0110	86	56	0101 0110	150	96	1001 0110	214	D6	1101 0110
23	17	0001 0111	87	57	0101 0111	151	97	1001 0111	215	D7	1101 0111
24	18	0001 1000	88	58	0101 1000	152	98	1001 1000	216	D8	1101 1000
25	19	0001 1001	89	59	0101 1001	153	99	1001 1001	217	D9	1101 1001
26	1A	0001 1010	90	5A	0101 1010	154	9A	1001 1010	218	DA	1101 1010
27	1B	0001 1011	91	5B	0101 1011	155	9B	1001 1011	219	DB	1101 1011
28	1C	0001 1100	92	5C	0101 1100	156	9C	1001 1100	220	DC	1101 1100
29	1D	0001 1101	93	5D	0101 1101	157	9D	1001 1101	221	DD	1101 1101
30	1E	0001 1110	94	5E	0101 1110	158	9E	1001 1110	222	DE	1101 1110
31	1F	0001 1111	95	5F	0101 1111	159	9F	1001 1111	223	DF	1101 1111
32	20	0010 0000	96	60	0110 0000	160	A0	1010 0000	224	E0	1110 0000
33	21	0010 0001	97	61	0110 0001	161	A1	1010 0001	225	E1	1110 0001
34	22	0010 0010	98	62	0110 0010	162	A2	1010 0010	226	E2	1110 0010
35	23	0010 0011	99	63	0110 0011	163	A3	1010 0011	227	E3	1110 0011
36	24	0010 0100	100	64	0110 0100	164	A4	1010 0100	228	E4	1110 0100
37	25	0010 0101	101	65	0110 0101	165	A5	1010 0101	229	E5	1110 0101
38	26	0010 0110	102	66	0110 0110	166	A6	1010 0110	230	E6	1110 0110
39	27	0010 0111	103	67	0110 0111	167	A7	1010 0111	231	E7	1110 0111
40	28	0010 1000	104	68	0110 1000	168	A8	1010 1000	232	E8	1110 1000
41	29	0010 1001	105	69	0110 1001	169	A9	1010 1001	233	E9	1110 1001
42	2A	0010 1010	106	6A	0110 1010	170	AA	1010 1010	234	EA	1110 1010
43	2B	0010 1011	107	6B	0110 1011	171	AB	1010 1011	235	EB	1110 1011
44	2C	0010 1100	108	6C	0110 1100	172	AC	1010 1100	236	EC	1110 1100
45	2D	0010 1101	109	6D	0110 1101	173	AD	1010 1101	237	ED	1110 1101
46	2E	0010 1110	110	6E	0110 1110	174	AE	1010 1110	238	EE	1110 1110
47	2F	0010 1111	111	6F	0110 1111	175	AF	1010 1111	239	EF	1110 1111
48	30	0011 0000	112	70	0111 0000	176	B0	1011 0000	240	F0	1111 0000
49	31	0011 0001	113	71	0111 0001	177	B1	1011 0001	241	F1	1111 0001
50	32	0011 0010	114	72	0111 0010	178	B2	1011 0010	242	F2	1111 0010
51	33	0011 0011	115	73	0111 0011	179	B3	1011 0011	243	F3	1111 0011
52	34	0011 0100	116	74	0111 0100	180	B4	1011 0100	244	F4	1111 0100
53	35	0011 0101	117	75	0111 0101	181	B5	1011 0101	245	F5	1111 0101
54	36	0011 0110	118	76	0111 0110	182	B6	1011 0110	246	F6	1111 0110
55	37	0011 0111	119	77	0111 0111	183	B7	1011 0111	247	F7	1111 0111
56	38	0011 1000	120	78	0111 1000	184	B8	1011 1000	248	F8	1111 1000
57	39	0011 1001	121	79	0111 1001	185	B9	1011 1001	249	F9	1111 1001
58	3A	0011 1010	122	7A	0111 1010	186	BA	1011 1010	250	FA	1111 1010
59	3B	0011 1011	123	7B	0111 1011	187	BB	1011 1011	251	FB	1111 1011
60	3C	0011 1100	124	7C	0111 1100	188	BC	1011 1100	252	FC	1111 1100
61	3D	0011 1101	125	7D	0111 1101	189	BD	1011 1101	253	FD	1111 1101
62	3E	0011 1110	126	7E	0111 1110	190	BE	1011 1110	254	FE	1111 1110
63	3F	0011 1111	127	7F	0111 1111	191	BF	1011 1111	255	FF	1111 1111

HEX ADDITION

HEXADECIMAL ADDITION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

HEX	F + 8	=	17	
HEX	10	=	16	DEC
HEX	7	=	7	DEC
HEX	17	=	23	DEC

HEX MULTIPLY

HEXADECIMAL MULTIPLICATION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

HEX	9 x 8	=	48	
HEX	40	=	64	DEC
HEX	8	=	8	DEC
HEX	48	=	72	DEC

Appendix F

ASSEMBLER ERROR CODES

The following error code numbers signify the TEKTRONIX Assembler error messages describing them. For 8002 μProcessor Labs purchased with 16k-byte program memory modules, error codes are displayed. For systems purchased with program memory modules totaling more than 16k-bytes, error messages are displayed along with the error codes. Upon assembly and in assembler listings, error codes and messages appear immediately below the source line containing an error.

***** ERROR: 001 (no message displayed.)

Indicates that a user entered WARNING message has assembled. Refer to WARNING directive explanation in Section 4.

***** ERROR: 002 SYMBOL ALREADY DEFINED

Indicates that the symbol defined has been previously defined in the program assembling sequence. Occurs when the same symbol is equated to two values (with EQU directive) or when the same symbol labels two instructions.

***** ERROR: 003 SYMBOL VALUE PHASE ERROR

Indicates that the label or EQU symbol value differs between passes, or that the section assignment of a label or EQU symbol value differs between passes.

***** ERROR: 004 ILLEGAL EQU OF GLOBALS

Indicates that an unbound global is assigned the value of another unbound global (with EQU directive). Error occurs because unbound globals are not assigned values in the current assembly.

***** ERROR: 005 GLOBAL DEFINITION MAY NOT USE HI, LO, OR ENDOF

Indicates that the value assigned to the global symbol involved HI, LO, or ENDOF function usage. Occurs when a global symbol is equated to HI(x) or LO(x), where x is an address, or ENDOF(y), where y is the section name whose ending address is to be found.

***** ERROR: 006 STRING EXPRESSION REQUIRED

Indicates that a numeric value appears where a string value is required. Operations requiring string expressions involve concatenation, SEG and NCHR function usage, and ASCII, TITLE, or STITLE directive usage.

***** ERROR: 007 UNDEFINED BLOCK OR ORG EXPRESSION

The operand expression of an ORG or BLOCK directive is either undefined or a forward reference. Occurs when an undefined or misspelled symbol appears in an ORG or BLOCK directive, or a symbol is assigned a value after the ORG or BLOCK references the symbol.

***** ERROR: 008 INVALID ORG OUT OF SECTION

Indicates that the ORG operand expression represents an address defined outside the current section. Examine previous RESUME or SECTION statements for errors.

***** ERROR: 009 NEGATIVE BLOCK LENGTH

Indicates that the BLOCK operand expression represents a negative value.

***** ERROR: 010 MACRO ALREADY DEFINED

Indicates that more than one MACRO directive contains the same name.

***** ERROR: 011 MACRO DEFINITION PHASE ERROR

Indicates two possible errors: The macro was called before being defined, or the macro was defined during the second assembler pass, but not the first.

***** ERROR: Ø12 MEMORY FULL ON MACRO CALL

Indicates insufficient space to perform macro expansion. Occurs when too many long arguments are specified for parameter substitution, too many symbols are entered in macro definition, or the macro repeats itself infinitely.

***** ERROR: Ø13 MISSING ENDR OR ENDIF

Indicates that a conditional assembly (IF or REPEAT) block failed to complete assembly. Occurs when a conditional assembly block begins assembly within a macro definition and the macro terminates (with an ENDM directive) before the conditional assembly terminates (with an ENDR or ENDIF directive).

***** ERROR: Ø14 DUPLICATE DEFINITION OF SECTION NAME

Indicates that the section name has already been defined as a label symbol during the current assembler pass.

***** ERROR: Ø15 END WITHIN INCLUDE FILE

Indicates that an END directive is present in an INCLUDE file.

***** ERROR: Ø16 ENDR OR ENDIF MIS-MATCHED

Indicates that an improper termination directive was used for a conditional assembly block. Occurs when ENDR is entered to terminate an IF block, ENDIF is entered to terminate a REPEAT block, or when IF and REPEAT blocks overlap each other producing the same effect.

***** ERROR: Ø17 ITERATION LIMIT EXCEEDED

Indicates an attempt to assemble a REPEAT block more than the specified number of times. If the allowed number of repeat cycles is left unspecified, the error message is displayed when 256 repeat cycles are completed.

***** ERROR: Ø18 MISPLACED ELSE

Indicates that an ELSE directive occurs outside its corresponding IF-ENDIF block, or that more than one ELSE directive occurs within the scope of one IF-ENDIF block.

***** ERROR: Ø19 OPERATION INVALID FOR ADDRESSES

Indicates that an operation allowing only scalar values was applied to an address value.

***** ERROR: Ø2Ø DIVISOR IS ZERO

Indicates that the Assembler attempted to divide by zero. Also occurs when the Assembler attempts to determine the remainder of a division by zero with the MOD operator (for example, A MOD Ø).

***** ERROR: Ø21 HI OR LO ALREADY APPLIED

Indicates an attempt to extract the most- or least-significant byte of an address expression, when the expression is already the result of a previous extraction.

***** ERROR: Ø22 ENDOF OPERAND IS SCALAR

Indicates that the specified section name in the ENDOF statement is a non-global, scalar symbol.

***** ERROR: Ø23 ENDOF ALREADY APPLIED

Indicates an attempt to perform an ENDOF function upon an address resulting from a previous ENDOF function.

***** ERROR: Ø24 ENDOF OPERAND IS NOT GLOBAL

Indicates that the specified section name in the ENDOF statement represents a non-global symbol.

***** ERROR: Ø25 OPERATION ON HI OR LO OF ADDRESS

Indicates an attempt to perform an arithmetic or unary operation upon an address that has had HI or LO applied to it.

***** ERROR: Ø26 ADDITION OF ADDRESSES

Indicates an attempt to add one address to another.

***** ERROR: Ø27 CONFLICTING SECTION BASES

Indicates an attempt to subtract or compare addresses based to different sections or having different ending byte addresses.

***** ERROR: Ø28 ADDRESS SUBTRACTED FROM SCALAR

Indicates an attempt to subtract an address from a scalar value.

***** ERROR: Ø29 NEGATIVE STRING LENGTH

Indicates that a negative value was specified for the string length when the string was declared with the STRING directive.

***** ERROR: Ø3Ø STRING LENGTH PHASE ERROR

Indicates that the string expression value differs between the assembler's first and second pass. Occurs when the string length expression contains a forward reference.

***** ERROR: Ø31 REDECLARATION OF STRING VARIABLE

Indicates a second attempt to declare the same string variable.

***** ERROR: Ø32 STRING DECLARATION PHASE ERROR

Indicates that the string value was defined during the assembler's second pass, but not its first.

***** ERROR: Ø33 INVALID STRING NAME

Indicates that an invalid string variable name has been entered as an operand in the STRING directive.

***** ERROR: Ø34 END INSIDE AN UNCLOSED BLOCK

Indicates that an END statement occurs within an IF, REPEAT, or MACRO definition block. Occurs when an ENDIF, ENDR, or ENDM directive is either missing or misspelled.

***** ERROR: Ø35 VALUE TRUNCATED TO BYTE

Indicates that the value entered exceeds one byte (value falls outside the range—128 to 255). The value is truncated to fall within one-byte range.

***** ERROR: Ø36 COLON FOLLOWS LABEL

Indicates that a colon (:), rather than a space, was encountered following a label. Occurs when a program is improperly converted to Tektronix Assembler format, or when the label is improperly entered.

***** ERROR: Ø37 EXTRA OPERANDS IGNORED

Indicates that extra operands appear in the statement. The complete statement entered prior to the extra operands is assembled, and the extra operands are ignored. Occurs when a statement is miscoded, an invalid delimiter occurs in the operand list, or a semicolon does not precede a comment. This error also occurs when a logical not "\" operator or a function follows what could be interpreted as a complete expression. This complete expression is either composed of or ends in a constant, a symbol, or a right parentheses ")". The portion of the statement that precedes the logical not operator or function is assembled and the remaining portion of the operand is ignored.

***** ERROR: Ø38 SET SYMBOL USED AS LABEL

Indicates that a symbol that was previously assigned a value with the SET directive appears in the label field of an instruction. Occurs when a SET symbol is redefined as a label, or when a subsequent SET operation is misspelled when reassigning a value to a symbol.

***** ERROR: Ø39 INVALID OPERATION CODE

Indicates that the Assembler is unable to recognize the operation in the statement, or that the Assembler disallows the operation to be processed in its entered context. Occurs when the operation is misspelled, an invalid delimiter follows the label, or a macro is called prior to its definition.

***** ERROR: Ø40 INVALID CHARACTER

Indicates that the Assembler encountered a character, outside the valid character set, that was not enclosed within double quotes.

***** ERROR: Ø41 SYNTAX ERROR

Indicates that the statement does not conform to the required syntax. Refer to Appendices B and C for required syntax for Assembler directives and 9900 instructions.

***** ERROR: Ø42 INVALID OPTION SEPARATOR

Indicates that the Assembler encountered an invalid delimiter between listing control options in the LIST or NOLIST directive operand field. Occurs when spaces delimit the options where commas are required, or when an invalid listing control option is entered.

***** ERROR: Ø43 NO LABEL ON EQU OR SET

Indicates that a symbol is either missing from or invalid for the label field of an EQU or SET directive.

***** ERROR: Ø44 INVALID MACRO NAME

Indicates that the macro name is missing from the operand field of the MACRO directive or that the macro name is an invalid symbol. Occurs when a previously defined symbol is entered as a macro name, a macro name is missing from the macro directive operand field, or an invalid delimiter is entered between the macro operation and macro name.

***** ERROR: Ø45 INVALID RELOCATION OPTION

Indicates an attempt to specify an invalid relocation option (other than ABSOLUTE) when declaring a section. When this error occurs, the Assembler ignores the invalid option, and handles the specified section as if it were byte-relocatable.

***** ERROR: Ø46 MACRO WITHIN A MACRO

Indicates that a macro definition statement was encountered within a macro expansion or a macro definition block.

***** ERROR: Ø47 INVALID EXCEPT IN MACRO

Indicates that an EXITM, ENDM, REPEAT, or ENDR directive appeared outside a macro definition block.

***** ERROR: Ø48 INVALID OPERAND

Indicates that the specified operand is either incomplete or inaccurate for the context required by the operation. If the required operand is an expression, this error indicates that the first item in the operand field is not a variable, constant, a left parentheses "(", a minus sign "–", or a logical not "\".

***** ERROR: Ø49 ADDRESS ASSIGNED TO STRING

Indicates an attempt to assign an address value to a string variable symbol.

***** ERROR: Ø5Ø SECTION DEFINITION PHASE ERROR

Indicates that the specified section or relocation option differs between the Assembler's first and second pass.

***** ERROR: Ø51 SECTION DEFINITION PHASE ERROR

Indicates that the specified section was defined during the second, but not the first, Assembler pass.

***** ERROR: Ø52 TOO MANY SECTIONS AND/OR GLOBALS

Indicates that the number of declared sections and global symbols exceeds 254. The Assembler does not accept the current section or global declaration.

***** ERROR: Ø53 INVALID RELOCATION OPTION

Indicates that the ABSOLUTE relocation option was specified in the RESERVE directive operand field.

***** ERROR: Ø54 NEGATIVE RESERVE LENGTH

Indicates that a negative-valued byte length was specified as the RESERVE operand expression.

***** ERROR: Ø55 INVALID SECTION NAME

Indicates that an invalid symbol was declared as a SECTION, COMMON, or RESERVE name. Occurs when the symbol name is misspelled, contains invalid characters, is a reserved word, or is a previously defined label.

***** ERROR: Ø56 INVALID RESERVE LENGTH

Indicates that the required RESERVE operand expression (specifying the number of bytes reserved for the current object module) is either entered incorrectly, entered without a comma before the expression, or absent from the RESERVE directive.

***** ERROR: Ø57 RESUME SECTION UNDEFINED

Indicates that the resumed section is defined in a later statement in the assembly process.

***** ERROR: Ø58 RESUME OF RESERVE SECTION

Indicates an attempt to resume a reserved section.

***** ERROR: 059 RESUMED SECTION INVALID

Indicates that the resumed section was declared after the 254th section or global symbol was declared.

***** ERROR: 060 GLOBAL OPERAND ALREADY DEFINED

Indicates that the global symbol was referenced before it was declared to be global. See GLOBAL directive explanation in Section 4.

***** ERROR: 061 GLOBAL DECLARATION PHASE ERROR

Indicates that a symbol was not declared global in both passes of the assembler.

***** ERROR: 062 TOO MANY SECTIONS AND GLOBALS

Indicates undefined globals, or more than 254 globals and sections defined.

***** ERROR: 063 INVALID RADIX

Indicates an invalid radix character in the constant. The 8002 μProcessor Lab Assembler recognizes only (H) hexadecimal, (O) or (Q) octal, and (B) binary radix codes.

***** ERROR: 064 INVALID DIGIT

Indicates an invalid digit in the indicated number base. For example, 10031B contains an invalid digit. Radix B indicates this to be a binary number, making digit 3 invalid.

***** ERROR: 065 UNMATCHED STRING OR PARAMETER DELIMITER

Indicates an unmatched quotation mark delimiter or square bracket delimiter.

***** ERROR: 066 LINE TOO LONG AFTER REPLACEMENT

Indicates expanded line is too long. Only 128 characters are allowed.

***** ERROR: Ø67 EXTRA REPLACEMENT IDENTIFIER

Indicates extra information following the replacement indicator in a macro definition block.

***** ERROR: Ø68 REPLACEMENT INVALID OUTSIDE OF MACRO

Indicates improper use of replacement indicators #, @, and % outside of a macro definition block.

***** ERROR: Ø69 UNDEFINED REPLACEMENT STRING

Indicates that the string variable has not yet been defined as a string.

***** ERROR: Ø70 INVALID REPLACEMENT IDENTIFIER

Indicates that the replacement specification used is invalid.

***** ERROR: Ø71 SCALAR VALUE REQUIRED

Indicates an address value where a scalar value was required.

***** ERROR: Ø72 INVALID EXPRESSION

Indicates that the specified expression is either incomplete or inaccurate for the context required by the operation. Expressions are recognizable when the following values appear in the first item position of the operand: a variable, a constant, a left parentheses "(", a minus sign "–", or a logical not character "\".

***** ERROR: Ø73 SECTION SIZE PHASE ERROR

Indicates that the number of bytes generated for this section during the first pass is smaller than the number of bytes generated during the second pass.

***** ERROR: Ø74 UNDEFINED SYMBOL

Indicates that a symbol in an expression has no value.

***** ERROR: Ø75 STRING TRUNCATED

Indicates that the number of characters assigned to the string is greater than the string definiton. See SET Strings, Section 2.

***** ERROR: Ø76 NEGATIVE SEG OPERAND

Indicates a negative number in the operand of the SEG function. See SEG, Section 2.

***** ERROR: Ø77 SEG STARTING OPERAND IS ZERO

Indicates a zero in the starting position of the SEG operand. See SEG, Section 2.

***** ERROR: Ø78 INSUFFICIENT WORKSPACE

Indicates that a temporary data manipulation area has been exceeded. Could be caused by conditional assembly or string functions that leave too little memory to perform the required operations.

***** ERROR: Ø79 VALUE TOO LARGE

Indicates that the space directive's operand value exceeds 255.

***** ERROR: Ø8Ø INVALID NAME SYMBOL

Indicates that the symbol in the operand field of the NAME directive begins with a non-alphabetic character and is, therefore, invalid.

***** ERROR: Ø81 ILLEGALLY SUBSTITUTED ENDM

Indicates that an ENDM directive was substituted within the body of a macro expansion before the normal end of the macro is encountered.

***** ERROR: Ø82 NESTED INCLUDE DIRECTIVE

Indicates that the file inserted into the program with the INCLUDE directive contains another INCLUDE directive.

***** ERROR: Ø83 MISSING ENDIF

Indicates that a conditional IF block with a missing ENDIF directive was included in the program.

***** ERROR: Ø84 MISSING ENDM FOR INCLUDED MACRO

Indicates that a macro definition block with a missing ENDM directive was included in the program.

***** ERROR: Ø85 STRING VALUE TOO LARGE

Indicates that a string value to be used as a number exceeds two characters in length.

***** ERROR: Ø86 SHIFT COUNT EXCEEDS 16

Indicates an attempt to shift right or left more than 16 bits.

***** ERROR: Ø87 TOO MANY SYMBOLS

Indicates a lack of room in the Assembler's symbol table to contain all symbols used by the program. The Assembler discontinues processing the program.

***** ERROR: Ø88 INVALID TRANSFER LABEL

Indicates that a label used for the transfer address on an END directive is an unbound global, a scalar, or the result of a previous HI, LO, or ENDOF function.

The following error messages apply only to the 9900.

***** ERROR: 254 REGISTER ADDRESS IS OUTSIDE OF CURRENT WORKSPACE

Indicates that an address specified in the register field of an instruction was in the same section as the current workspace, but was more than 16 words -- beyond the base of the workspace.

***** ERROR: 253 REGISTER REFERENCED AT AN ODD ADDRESS

Indicates that an odd address was specified as the location of a workspace register.

***** ERROR: 252 REGISTER ADDRESS IS NOT IN SAME SECTION AS WORKSPACE

Indicates that an address specified in a register field of an instruction was not in the same section as the current workspace.

***** ERROR: 251 SCALAR REGISTER SPECIFICATION NOT IN THE RANGE 0–15

Indicates that a scalar value greater than 15 was specified in a register field.

***** ERROR: 250 RØ NOT VALID AS AN INDEX REGISTER

Indicates that a register Ø was specified as an index register.

***** ERROR: 249 SYMBOLIC MODE ASSUMED FOR ODD WORKSPACE ADDRESS

Indicates that an odd address within the current workspace was specified.

***** ERROR: 248 SCALAR OPERAND INVALID FOR WPNT

Indicates that an invalid scalar operand was specified for WPNT.

***** ERROR: 247 WPNT OPERAND CONTAINS A FORWARD REFERENCE

Indicates a forward reference in the operand field of a WPNT instruction.

***** ERROR: 246 ODD ADDRESS NOT VALID AS OPERAND TO WPNT

Indicates an invalid odd address in the operand field of WPNT instruction.

***** ERROR: 245 RELATIVE JUMP OR CRU OFFSET OUT OF RANGE

Indicates that either a relative branch or a CRU offset is out of range.

***** ERROR: 244 JUMP OUT OF CURRENT SECTION

Indicates that the destination of a relative jump instruction is outside the current section.

***** ERROR: 243 JUMP TO AN ODD ADDRESS

Indicates that the destination of jump instruction is an odd address.

***** ERROR: 242 COUNT IS TOO LARGE

Indicates that the shift count, CRU transfer length, or XOP code is greater than 15.

***** ERROR: 241 INVALID MACHINE INSTRUCTION OPERAND

Indicates that the syntax of the machine instruction operand is invalid.

***** ERROR: 240 INVALID WORD ALIGNMENT

Indicates that a word value was generated at an odd address.

***** ERROR: 239 WPNT OPERAND INVOLVES HI, LO, OR ENDOF

Indicates that the operand to WPNT has had a HI, LO, or ENDOF function applied to it.

Appendix G
RESERVED WORDS

The 9900 Microprocessor instruction mnemonics, register symbols, and Tektronix Assembler directive names must not be used as symbolic labels. The following names are reserved for these special uses:

9900 INSTRUCTION MNEMONICS

A	C	DEC	JGT	JNE	LWPI	RT	SOC	SWPB
AB	CB	DECT	JH	JNO	MOV	RTWP	SOCB	SZC
ABS	CI	DIV	JHE	JOC	MOVB	S	SRA	SZCB
AI	CKOF	IDLE	JL	JOP	MPY	SB	SRC	TB
ANDI	CKON	INC	JLE	LDCR	NEG	SBO	SRL	X
B	CLR	INCT	JLT	LI	NOP	SBZ	STCR	XOP
BL	COC	INV	JMP	LIMI	ORI	SETO	STST	XOR
BLWP	CZC	JEQ	JNC	LREX	RSET	SLA	STWP	

9900 REGISTER SYMBOLS

RØ	R5	R1Ø	R15
R1	R6	R11	
R2	R7	R12	
R3	R8	R13	
R4	R9	R14	

RESERVED FOR FUTURE USE

XREF

TEKTRONIX ASSEMBLER DIRECTIVES, OPTIONS & OPERATORS

ABSOLUTE	END	INCLUDE	ORG	SHR
ASCII	ENDIF	LIST	PAGED	SPACE
BASE	ENDM	LO	REPEAT	STITLE
BLOCK	ENDOF	MACRO	RESERVE	STRING
BYTE	ENDR	ME	RESUME	SYM
CND	EQU	MEG	SCALAR	TITLE
COMMON	EXITM	MOD	SECTION	TRM
CON	GLOBAL	NAME	SEG	WARNING
DEF	HI	NCHR	SET	WORD
ELSE	IF	NOLIST	SHL	WPNT

TEKTRONIX®
committed to technical excellence

MANUAL CHANGE INFORMATION

PRODUCT 9900 ASSEMBLER & EMULATOR USER'S

CHANGE REFERENCE C1/178

DATE 1-27-78

CHANGE:	DESCRIPTION

070-2417-00

The 8002 μProcessor Lab Hardware Test Manual, mentioned in the Documentation Overview section, is no longer available for distribution.

Page 2-5 – The operand syntax for the fifth operand type ("A 16-bit indexed memory address . . .") should be as follows:

expression (R1)
@expression (R1)
.
.
expression (R15)
@expression (R15)
expression (expression)
@expression (expression)

Page 4-26 – The syntax line for the WPNT directive should be as follows:

[symbol] WPNT {expression} [;charstring]

Page 4-50 – The operand ØF7H should follow the WORD directive in the example.

Page 4-55 – The syntax line for the SECTION directive should be as follows:

[symbol] SECTION {symbol} [,ABSOLUTE] [;charstring]

Page 4-57 – The syntax line for the COMMON directive should be as follows:

[symbol] COMMON {symbol} [,ABSOLUTE] [;charstring]

Page 9-9 – The sixteenth line ("Valid relocation types are PAGE, INPAGE, and BYTE.") should be removed.

Page 9-13 – The last line on the page should be as follows:

program in this example, the appropriate command would be "GO 4Ø".

Page 9-17 – Line 21 ("is specified, a transfer of Ø is generated.") should be as follows:

"is specified, a transfer address of Ø is generated."

CHANGE:	DESCRIPTION

Page C-2 – The explanation of operand notation r should be as follows:
one of the workspace registers, RØ to R15; or an expression that evaluates to a scalar value of Ø to 15; or an even address not greater than 30 bytes beyond the current workspace base as defined by WPNT.

Appendix F: The following additions and corrections should be noted.

***** ERROR: Ø15 END DIRECTIVE NOT VALID WITHIN AN INCLUDE FILE

Indicates that an END directive is present in an INCLUDE file.

***** ERROR: Ø19 OPERATION INVALID FOR ADDRESS

Indicates that an operation allowing only scalar values was applied to an address value.

***** ERROR: Ø21 TEXT FOLLOWING "]" IGNORED

Indicates that information following a bracketed macro parameter has been ignored.

***** ERROR: Ø36 INVALID CHARACTER FOLLOWS LABEL

Indicates that a character other than a space was encountered following a lable.

***** ERROR: Ø38 STRING VARIABLE USED AS LABEL

Indicates that a string variable is present in the label field of an instruction. Label is ignored.

***** ERROR: Ø42 INVALID OPTION OR SEPARATOR

Indicates that the Assembler encountered an invalid delimiter between listing control options in the LIST or NOLIST directive operand field. Occurs when spaces delimit the options where commas are required, or when an invalid listing control option is entered.

***** ERROR: Ø52 TOO MANY SECTIONS OR GLOBALS

Indicates that the number of declared sections and global symbols exceeds 254. The Assembler does not accept the current section or global declaration.

CHANGE:	DESCRIPTION

***** ERROR: Ø9Ø ENDOF APPLIED TO A BOUND GLOBAL

Indicates that an ENDOF function was used with a bound GLOBAL instead of a SECTION. In the case of an unbound GLOBAL, the function will be resolved at link time.

***** ERROR: Ø91 UNABLE TO ASSIGN INCLUDE FILE

Indicates that TEKDOS could not gain access to the file. This message will be accompanied by a message on the console during each pass. An SRB status code will indicate the reason that TEKDOS could not assign the file.